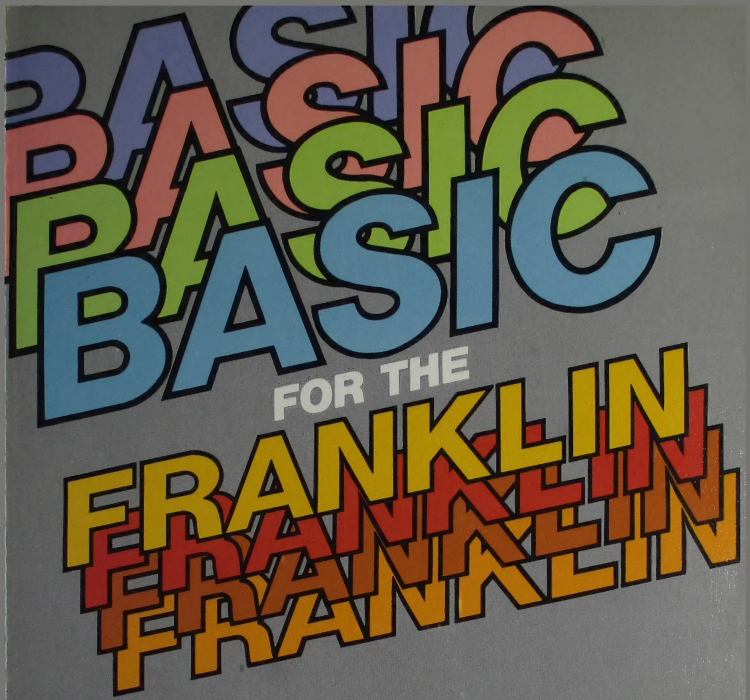


PROGRAMMING and APPLICATIONS

SAL MANETTA
LARRY JOEL GOLDSTEIN
MARTIN GOLDSTEIN

FRANKLIN COMPUTER

BASIC for the Franklin
Programming and Applications

Sal Manetta

Larry Joel Goldstein

Martin Goldstein

Robert J. Brady Company
A Prentice-Hall Publishing
and Communications Company
Bowie, MD 20715

BASIC for the Franklin: Programming and Applications

Library of Congress Cataloging in Publication Data

Manetta, Sal.
BASIC for the Franklin.

Includes index.
1. Franklin computer—Programming. 2. Basic (Computer program language) I. Goldstein, Larry Joel. II. Goldstein, Martin, 1919 Mar. 28- III. Title.
QA76.8.F7M36 1983 001.64'2 83-6382
ISBN 0-89303-341-3

Prentice-Hall International, Inc., London
Prentice-Hall Canada, Inc., Scarborough, Ontario
Prentice-Hall of Australia, Pty., Ltd., Sydney
Prentice-Hall of India Private Limited, New Delhi
Prentice-Hall of Japan, Inc., Tokyo
Prentice-Hall of Southeast Asia Pte. Ltd., Singapore
Whitehall Books, Limited, Petone, New Zealand
Editora Prentice-Hall Do Brasil LTDA., Rio de Janeiro

Printed in the United States of America

83 84 85 86 87 88 89 90 91 92 93 10 9 8 7 6 5 4 3 2 1

Executive Editor: Terrell Anderson
Production Editor: Michael J. Rogers
Assistant Art Director: Bernard Vervin
Brady Cover Design: Don Sellers
Franklin Cover Design: Frank Williams
Typesetting by: VITRO Laboratories, a division of Automated Industries, Inc. Silver Spring, MD
Printed by: R.R. Donnelley & Sons, Harrisonburg, VA
Indexer: Leah Kramer

CONTENTS

1
A Brief Look At Computers

1.1 Introduction

As is the case when you're learning anything else that's new and different, you should begin at the right place and avoid shortcuts, at least initially. If you want to learn something about personal computers in general, start with the Franklin publication *Playing With a Full Deck,* available at many computer stores. Then, assuming you've bought a Franklin computer, set it up according to the directions provided in the *User Reference Manual* that comes with each machine. Be sure to work your way through the entire manual so that you're completely comfortable with all aspects of operating the system you bought.

Next, buy a few applications programs to get a feel for what programs are and how they work. There are literally tens of thousands of programs on the market for the Franklin. You'll probably want to take advantage of this vast library before you try any programming yourself.

Let's say that you've done all of this and now you're interested in learning how to do some of your own programming. Look no further! You're in the right place. This book takes you through the fundamentals of talking with your computer in Floating Point BASIC. Throughout, you'll find exercises that test your understanding of the material and suggest programs that you can write. Many of the exercises are designed to give you an insight into how computers are used in business and industry. There are even a number of applications for your home, and, for good measure, opportunities to build a few computer games.

The Franklin Computer

Although the first "personal" computers appeared on the market only a few years ago, they are incredibly sophisticated devices, using many of the same

features as the large main-frames. To program, you must know a little more about some of the features of most computers than was mentioned in the *User Reference Manual.* Some of what follows may be a review for you, some of it new.

1.2 What is a Computer?

At the heart of every Franklin is a little device called a microprocessor. It acts as the central processing unit (or CPU) to perform whatever commands you give to the computer. This unit carries out arithmetic and all logical operations in the machine.

There's also the input unit that lets you send information to the computer and an output unit that allows the computer to send information to you. The keyboard is the input device on the Franklin, and the output device is usually the monitor, a printer for paper copy, or both.

In addition, there's a memory that lets the computer "remember" numbers, words, and paragraphs, as well as the list of commands you want the computer to perform. Every Franklin has two kinds of memory, RAM and ROM, each with its own advantages and disadvantages. **ROM** stands for "read only memory," meaning that only the CPU can read it and that you cannot record anything in it. ROM contains the information the computer needs to understand your commands.

RAM stands for "random access memory." When you type characters on the keyboard, they're stored in RAM; similarly, RAM also stores the results of your calculations awaiting output to you. But, as you probably know, RAM is erased when you turn the computer off, so it can't be used to store data in a permanent form. Nevertheless, RAM is used as the computer's main working storage because it takes only about a millionth of a second to store or retrieve a piece of data from RAM.

Diskettes, despite their floppy natures, provide a permanent storage medium, each capable of sorting several hundred thousand characters. To give you some idea of what this represents, realize that a double-spaced typed page contains about 3,000 characters.

Figure 1-1 shows how the different components of a computer system relate to one another.

1.3 Meet Your Franklin Computer

As you probably remember from working with the *User Reference Manual,* the best—perhaps the only way—to master the operations of your computer is to sit down at the keyboard and verify each statement as you read it.

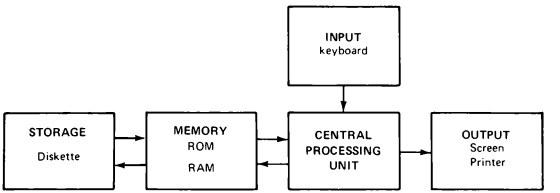


Figure 1-1. The main components of a computer.

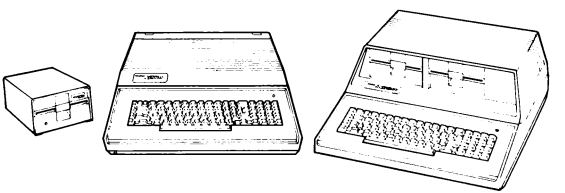


Figure 1-2. The Franklin computer.

By way of review, take a look at your keyboard. In most ways, the Franklin keyboard works exactly like a standard typewriter. If you type characters without depressing the SHIFT key, you'll get lower case. You can program in Floating Point BASIC using lower case characters. When you simultaneously depress the SHIFT key and a character key, you'll get an upper case character.

Unlike a standard typewriter, however, the Franklin has no SHIFT LOCK function. Instead, all Franklins have an ALPHA (short for alphabet) LOCK, which is slightly different.

In a typewriter's SHIFT LOCK mode, all the characters on the keyboard shift to either an upper case letter or the top character on the punctuation keys. In the Franklin's ALPHA LOCK mode, only the alphabetic characters shift to upper case; the punctuation keys don't shift to their top characters.

To enter LOCK mode, hit the LOCK key. To get out of ALPHA LOCK, just hit the LOCK key again.

The group of twelve keys on the right of the keyboard is called the numeric pad. It consists of the ten digits, the "greater than" sign (>), and the decimal point. If you do a lot of numerical calculations or if you write reports with charts and numbers, you'll find using the numeric pad faster than using the

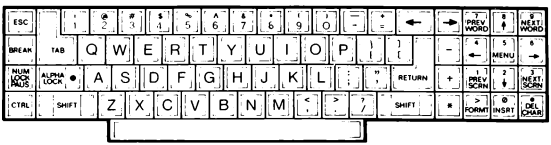


Figure 1-3. The Franklin computer keyboard.

numbers on the top row of keys. It'll also help you to avoid confusing the number 0 with the letter O.

The minus sign, the plus sign, and the asterisk are located just to the left of the numeric pad for easy access when you're using programs that process financial data.

Now turn on your computer and your monitor. The computer should display two symbols, the prompt and the cursor. This prompt is actually the Floating Point BASIC prompt and it means that the Franklin is ready to accept BASIC instructions.

You're already aware that the RETURN key moves the cursor to the next line to be typed. But the RETURN key has another function that relates particularly to programming. It signals the computer to accept the line you just typed. Until you hit RETURN, as far as the computer is concerned, whatever you may have typed doesn't exist.

Type anything that comes to mind until you are at the bottom of the screen. If you hit RETURN, the entire contents of the screen will move up by one line and the line at the top of the screen will disappear. This movement of lines on and off the screen is called **scrolling.**

As you may have already noticed, the computer will respond to some of your typed lines with error messages. Don't worry about these now. The computer has been taught to respond only to certain typed commands. If it encounters a command that it doesn't recognize, it will announce this fact with an error message. Don't be intimidated by the occasional slaps on the wrist handed out by your computer. Whatever happens, don't let these "slaps" stop you from experimenting. The worst that can happen is that you might have to turn your computer off and start all over!

By this time, your screen should look pretty cluttered. To clear it, type **HOME** and then press RETURN. All characters on the screen will be erased and only the cursor will remain. The cursor is positioned in the upper left corner of the screen, its so-called "home" position.

TEST YOUR UNDERSTANDING 1 (answers below)*

a. Type your name on the screen.
b. Erase the screen.

Unless you're a superb typist, you'll eventually make typing errors. Let's discover how to correct them. Type a few characters, but don't hit the RETURN key. Now hit the backspace key. (This is the key labelled ←.) Note that this key makes the cursor backspace, one space at a time, erasing the characters it passes over. You will not see the erasures on the screen, but they do occur in the computer's memory. This is another difference between a typewriter and a computer keyboard. Note, however, that you may use the backspace to correct lines only if they haven't been sent to the computer via the RETURN key.

There are other ways to correct typing errors, but for now let's be content with the methods discussed above.

Exercises

Type the following expressions on the screen. After each numbered exercise, clear the screen.

1. 10 HELLO! I'M YOUR NEW OWNER
2. 10 ARITHMETIC
3. 10 PRINT 3+7
4. 20 LET A=3-5
5. 20 5% of 68
6. 10 IF 38>-5
7. 10 X=5: PRINT X
8. 20 IF X.>0 THEN 50
9. 10 LET X=10
   20 LET Y=50.35
10. 200 Y=X*2-5
    300 PRINT Y,"Y"

ANSWERS TO TEST YOUR UNDERSTANDING 1

a. Type your name, ending the line with RETURN.
b. Type **HOME** followed by RETURN.

*Answers to the TEST YOUR UNDERSTANDING questions follow the exercises of each section.

2

Getting Started in Floating Point BASIC

2.1 Computer Languages and Programs

Just as humans use languages to communicate with one another, computers use languages to communicate with other electronic devices (such as printers), human operators, and other computers. There are hundreds of computer languages in use today. However, the most common one for microcomputers is called **BASIC** (Beginners All-purpose Symbolic Instructional Code). This is the best computer language to learn first since it is the most elementary computer language used by your Franklin. The first version of BASIC (not Floating Point) was developed especially for computer novices by John Kemeny and Thomas Kurtz at Dartmouth College. In the next few chapters, we will concentrate on learning the fundamentals of Floating Point BASIC. In the process, you'll learn a great deal about the way in which a computer may be used to solve problems.

Many people think of a computer as an "electronic brain" which somehow has the power of human thought. This is very far from the truth. The electronics of the computer and the rules of the Floating Point BASIC language allow it to recognize a very limited vocabulary, and to take various actions based on the data which is given to it. It's very important to recognize that the computer does not have "common sense." The computer will attempt to interpret whatever data you input. If your input is a recognizable command, the computer will perform it. It doesn't matter that the command makes no sense in a particular context. The computer has no way to make such judgments. It can

only do what you instruct it to do. Because of the computer's inflexibility in interpreting commands, you must tell the computer **exactly** what you want it to do. Don't worry about confusing the computer. If you provide a command in an incorrect form, you won't damage the machine in any way! In order to make the computer do what you want, it's necessary to learn its language.

Let's begin to learn something about Floating Point BASIC. Assume that you have followed the start-up instructions of the previous chapter and the computer shows that it is ready to accept further instructions by displaying the Floating Point BASIC prompt:

```
] ■
```

From this point on, a typical session with your computer might go like this:

1. Type in a series of instructions in Floating Point BASIC. Such a series of instructions is called a **program.**
2. Locate and correct any errors in the program.
3. Tell the computer to carry out the series of instructions in the program. This step is called **running the program.**
4. Obtain the output requested by the program.
5. Either: (a) run the program again; or (b) repeat steps 1-4 for a new program; or (c) end the programming session (turn off the computer and go have lunch).

To fully understand what is involved in these five steps, let's consider a particular example. Suppose that you want the computer to add 5 and 7. First, you would type the following instructions in either upper or lower case:

```
10 PRINT 5+7 [RETURN]
20 END [RETURN]
```

This sequence of two instructions constitutes a program to calculate 5 + 7. Note that as you type the program the computer records your instructions, but does not carry them out. As you're typing a program, the computer provides you with an opportunity to change, delete, and correct instruction lines. (More on how to do this later.) Once you're satisfied with your program, tell the computer to run it (that is, to execute the instructions) by typing the command:

```
RUN [RETURN]
```

The computer will run the program and display the answer:

```
12
```

If you wish the computer to run the program a second time, type **RUN** again and press [RETURN].

Running a program doesn't erase it from RAM. Therefore, if you wish to add instructions to the program or change the program, you may continue typing just as if the **RUN** command hadn't intervened. For example, if you

wish to include in your program the problem of calculating 5 - 7, you type the additional line

```
15 PRINT 5-7
```

To see the program currently in memory, type LIST (no line number), then hit the RETURN key. The program consists of the following three lines, now displayed on the screen:

```
10 PRINT 5+7
15 PRINT 5-7
20 END
```

Note how the computer puts line 15 in proper sequence. Type **RUN** RETURN again, and the computer will display the two answers:

```
12
-2
```

In the event that you now wish to go on to another program, type the command:

```
NEW
```

This erases the previous program from RAM and prepares the computer to accept a new program. You should always remember the following important fact:

RAM can contain only one program at a time.

TEST YOUR UNDERSTANDING 1 (answers on page 10)

a. Write and type in a BASIC program to calculate 12.1 + 98 + 5.32.
b. Run the program of a.
c. Erase the program of a. from RAM.
d. Write a program to calculate 48.75 - 1.674.
e. Type in and run the program of d.

Floating Point BASIC on the Franklin operates in two distinct modes. In **command mode,** the computer accepts typed program lines and commands (like **RUN** and **NEW**) used to manipulate programs. The computer identifies a program line by its line number. Program lines aren't immediately executed. Rather, they're stored in RAM until you tell the computer what to do with them. On the other hand, commands are executed as soon as they are given.

In the **execute mode,** the computer runs a program. In this mode, the screen is under control of the program.

When you turn on the computer it's automatically in command mode. The command mode is indicated by the presence of the] ■ prompt on the screen. The **RUN** command puts the computer into execute mode. After you run the

program the computer redisplays the] ■ prompt indicating that it's back in command mode.

The computer is a stern taskmaster! It has a very limited vocabulary (Floating Point BASIC) and this vocabulary must be used according to very specific rules concerning the order of words, punctuation, and so forth. However, Floating Point BASIC allows for some freedom of expression. For example, any extra spaces are ignored. Thus, Floating Point BASIC interprets all of the following instructions as identical:

```
10 PRINT A
10 PRINT      A
10     PRINT A
```

ANSWERS TO TEST YOUR UNDERSTANDING 1

a. 10 PRINT 12.1+98+5.32
   20 END
b. Type **RUN**
c. Type **NEW**
d. 10 PRINT 48.75 - 1.674
   20 END
e. Type in program followed by **RUN**.

2.2 Elementary Floating Point BASIC Programs

Keyboard characters are linked together to form the vocabulary of the Floating Point BASIC language. The simplest words in this vocabulary are the so-called constants.

Floating Point BASIC Constants

Floating Point BASIC allows you to manipulate numbers and text. The rules for manipulating numerical data differ from those for handling text, however. In Floating Point BASIC you distinguish between these two types of data as follows: a **numeric constant** is a number and a **string constant** is a sequence of keyboard characters, which may include letters, numbers, or any other keyboard symbols. The following are examples of numeric constants:

```
5, -2, 3.145, 23456, 456.78345676543987, 27134566543
```

The following are examples of string constants:

```
"John", "Accounts Receivable", "$234.45 Due" "Dec. 4,1981"
```

Note that string constants are always enclosed in quotation marks. In order to avoid vagueness, quotation marks may not appear as part of a string constant. (In practice, an apostrophe ' should be used as a substitute for " within a string constant.) Although numbers may appear within a string constant, you cannot use such numbers in arithmetic. Only numbers outside quotation marks may be used for arithmetic.

For certain applications, you may wish to specify your numeric constants in **exponential format.** This will be especially helpful in the case of very large and very small numbers. Consider the number 15,300,000,000. It's very inconvenient to type all the zeros. This large number can be written in handy shorthand as 1.53E10. The 1.53 indicates the first three digits of the number. The E10 means that you move the decimal point in the 1.53 to the **right** 10 places. Similarly, the number -237,000 may be written in the exponential format as -2.37E5. The exponential format may also be used for very small numbers. For example, the number .00000000054 may be written in exponential format as 5.4E-10. The -10 indicates that the decimal point in 5.4 is to be moved 10 places to the **left.**

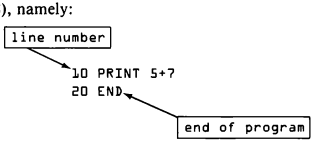
TEST YOUR UNDERSTANDING 1 (answers on page 18)

a. Write these numbers in exponential format: .00048, -1374.5
b. Write these numbers in decimal format: -9.7E3, 9.7E-3, -9.7E-3

There'll be more about constants later. For example, you'll find the number of digits of accuracy you can get, how to round off numbers, and so forth. Right now, you know more than enough to get started. So instead of concentrating on the fine points now, let's learn enough to make the computer **do something.**

Floating Point BASIC Programs

Let's look again at the Floating Point BASIC program in Section 2.1 (page 8), namely:



This program illustrates two very important features common to all Floating Point BASIC programs:

1. The instructions of a program must be numbered. Each line must start with a line number. The computer executes instructions in order of increasing line number.
2. The **END** instruction identifies the end of the program. On encountering this instruction, the computer stops running the program and displays] ■.

Line numbers need not be consecutive. For example, it's perfectly acceptable to have a program whose line numbers are 10, 23, 47, 55, or 100. Also note that it's not necessary to type instructions in their numerical order. You could type line 20 and then go back and type line 10. The computer will sort out the lines and rearrange them according to increasing number when you run the program. This feature is especially helpful in case you accidentally omit a line while typing your program.

Here is another important fact about line numbering. If you type two lines having the same line number, the computer erases the first version and remembers the second version. This feature is very useful for correcting errors: if a line has an error, just retype it!

Your Franklin will perform all the standard calculations that can be done with a calculator. Since most people are familiar with the operation of a calculator, let's start by writing programs to solve various arithmetic problems.

Most arithmetic operations are written in customary fashion. For example, addition and subtraction are written for the computer in the usual way:

```
5+4, 9 - 8
```

Multiplication, however, is typed using the symbol * which shares the 8 key. Therefore, the product of 5 and 3 is typed as:

```
5*3
```

Division is typed using /. Therefore, 8.2 divided by 15 is typed as:

```
8.2/15
```

Example 1. Write a Floating Point BASIC program to calculate the sum of 54.75, 78.83, and 548.

Solution. The sum is indicated by typing:

```
54.75+78.83+548
```

The Floating Point BASIC instruction for printing data on the screen is **PRINT**, so the program appears as:

```
10 PRINT 54.75+78.83+548
20 END
```

Floating Point BASIC carries out arithmetic operations in a special order. It scans an expression and carries out all multiplication and division operations first, **proceeding in left-to-right order.** It then returns to the left side of the expression and performs addition and subtraction, also in a left-to-right order. If parentheses occur, these are evaluated first following the same rules stated above. If parentheses occur within parentheses, the innermost parentheses are evaluated first.

Example 2. What are the numerical values which Floating Point BASIC will calculate from these expressions?

(a) (5+7)/2 (b) 5+7/2
(c) 5+7*3/2 (d) (5+7*3)/2

Solution.

(a) The computer first applies its rules for the order of calculation to determine the value in the parentheses, namely 12. It then divides 12 by 2 to obtain 6.

(b) The computer scans the expression from left to right performing all multiplication and division in the order encountered. First it divides 7 by 2 to obtain 3.5. It then rescans the line and performs all additions and subtractions in order. This gives

```
5+3.5=8.5
```

(c) The computer first performs all multiplication and division in order:

```
5+10.5
```

Now it performs addition and subtraction to obtain 15.5.

(d) The computer calculates the value of all parentheses first. In this case, it computes 5+7*3=26. (Note that it does the multiplication first!) Next it rescans the line, which now looks like:

```
26/2
```

It performs the division to obtain 13.

TEST YOUR UNDERSTANDING 2 (answer on page 18)

Calculate 5+3/2+2 and (5+3)/(2+2).

Example 3. Write a Floating Point BASIC program to calculate the quantity

$$\frac{22\times18+34\times11-12.5\times8}{27.8}$$

Solution. Here's the program:

```
10 PRINT (22*18+34*11-12.5*8)/27.8
20 END
```

Note the parentheses in line 10. They tell the computer that the entire quantity in parentheses is to be divided by 27.8. Without the parentheses, the computer would divide -12.5*8 by 27.8 and add 22*18 and 34*11 to the result.

TEST YOUR UNDERSTANDING 3 (answers on page 18)

Write Floating Point BASIC programs to calculate:

a. $((4\times3+5\times8+7\times9)/(7\times9+4\times3+8\times7))\times48.7$
b. 27.8 % of (112 + 38 + 42)
c. The average of the numbers 88, 78, 84, 49, 63

Printing Words

So far, the **PRINT** instruction has been used only to display the answers to numerical problems. However, this instruction is very versatile. It also allows you to display string constants. For example, consider this instruction:

```
10 PRINT "PATIENT HISTORY"
```

During program execution, this statement will create the following display:

```
PATIENT HISTORY
```

In order to display several string constants on the same line, separate them by commas in a single **PRINT** statement. For example, consider the instruction:

```
10 PRINT "AGE", "BIRTHPLACE"
```

After you execute the program you'll see:

```
AGE          BIRTHPLACE
```

Both numeric constants and string constants may be included in a single **PRINT** statement, for example

```
100 PRINT "AGE", 65.43
```

Here's how the computer determines the spacing on a line. Each line is divided into print zones. Two print zones consist of 16 spaces and one zone has 8 spaces. By placing a comma in a PRINT statement, you are telling the computer to start the next string of text at the beginning of the next print zone. Thus, for example, the two words above, AGE and BIRTHPLACE, begin in columns 1 and 17, respectively. (See Figure 2-1.) Note that the third print zone

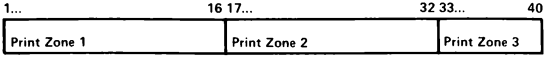


Figure 2-1. Print zones.

(positions 33-40) is available only if nothing is printed in positions 24-32. Moreover, to use print zone 2, you must leave print position 32 blank.

TEST YOUR UNDERSTANDING 4 (answer on page 18)

Write a program to print the following display.

```
                NAME
LAST            FIRST           MIDDLE
SMITH           JOHN            DAVID
```

Example 4. Suppose that a distributor of office supplies sells 50 chairs and 5 desks. The chairs cost $59.70 each and are subject to a 30 percent discount. The desks cost $247.90 each and are also subject to a 30 percent discount. Prepare a bill for the shipment.

Solution. Insert two headings on the bill: Item and Cost. Then print two lines, corresponding to the two types of items shipped. Finally, calculate the total due as shown here.

```
10 PRINT "ITEM","COST"
20 PRINT
30 PRINT "CHAIR",50*(59.7-.3*59.7)
40 PRINT "DESK", 5*(247.9-.3*247.9)
50 PRINT
60 PRINT "TOTAL DUE",50*(59.7-.3*59.7)
   +5*(247.9-.3*247.9)
70 END
```

Note the **PRINT** statements in lines 20 and 50. They specify that a blank line is to be printed. If you now type **RUN** (followed by RETURN), the screen will look like this:

```
ITEM            COST

CHAIR           2089.5
DESK            867.65

TOTAL DUE       2957.15
```

You may think that the above invoice is somewhat sloppy because the columns of figures are not properly aligned. Patience! You'll learn to align the columns after a bit more programming.

TEST YOUR UNDERSTANDING 5 (answer on page 18)

Write a computer program which creates the following display.

```
                    BUDGET-APRIL

FOOD                387.5
RENT                475.
CAR                 123.71
UTILITIES           146.
ENTERTAINMENT       100.
                    ------
TOTAL           (Calculate total)
```

Exponentiation

Suppose that A is a number and N is a positive whole number (this means that N is one of the numbers 1,2,3,4,...). Then A **raised to the Nth power** is the product of A times itself N times. This quantity is usually denoted in mathematics texts as A^N, and the process of calculating it is called **exponentiation.** For example,

```
2^3=2*2*2=8,  5^7=5*5*5*5*5*5*5=78125
```

It's possible to calculate A N by repeated multiplication. However, if N is large, the typing can be tiresome. Floating Point BASIC provides a shortcut for typing this function. Exponentiation is denoted by the symbol ^, which is produced by hitting the key with the upward-pointing arrow (this symbol shares the 6 key at the top of the keyboard). The operation of exponentiation takes precedence over multiplication and division. The following example illustrates the point.

Example 5. Determine the value which Floating Point BASIC assigns to this expression:

```
20*3-5*2^3
```

Solution. The exponentiation is performed first to yield:

```
20*3-5*8=60-40
        =20
```

TEST YOUR UNDERSTANDING 6 (answers on page 18)

Evaluate the following first manually and then by computer program.

a. 2^4x3^3
b. 2^2x3^3-12^2/3^2x2

Exercises (answers on page 195)

Write Floating Point BASIC programs to calculate the following quantities.

1. 57 + 23 + 48
2. 57.83x(48.27-12.54)
3. 127.86/38
4. 365/.005 + 1.02^5
5. Make a table of the first, second, and third powers of the numbers 2,3,4,5,and 6. Put all first powers in a column, all second powers in another column, and so forth.
6. Mrs. Anita Smith visited her doctor regarding a broken leg. Her bill consisted of $45 for removal of the cast, $35 for therapy, and $5 for drugs. Her major medical policy pays 80 percent directly to the doctor. Use the computer to prepare an invoice for Mrs. Smith.
7. A school board election is held to elect a representative for a district consisting of Wards 1,2,3, and 4. There are three candidates, Mr. Thacker, Ms. Hoving, and Mrs. Weatherby. The tallies by candidate and ward are as follows.

	Ward 1	Ward 2	Ward 3	Ward 4
Thacker	698	732	129	487
Hoving	148	928	246	201
Weatherby	379	1087	148	641

Write a Floating Point BASIC computer program to calculate the total number of votes for each candidate, as well as the total number of votes cast.

Describe the output from each of these programs.

8.
```
10 PRINT 8*2-3*(2^4-10)
20 END
```

9.
```
10 PRINT "SILVER","GOLD","COPPER"
20 PRINT 327,449,1052
30 END
```

10.
```
10 PRINT , "GROCERIES","MEATS"
20 PRINT "MON", "1,245","2,348"
30 PRINT "TUE", "  248","3,459"
40 END
```

Convert the following numbers to exponential format.

11. 23,000,000
12. 175.25
13. -200,000,000
14. .00014
15. -.000000000275
16. 53,420,000,000,000,000

Convert the following numbers in exponential format to standard format.

17. 1.59E5
18. -20.3456E6
19. -7.956E-12
20. 2.39456E-18

ANSWERS TO TEST YOUR UNDERSTANDING 1,2,3,4,5 and 6

1: a. 4.8E-4, -1.3745E3 b. -9700, .0097, -.0097

2: 8.5, 2

3: a.
```
10 PRINT ((4*3+5*8+7*9)/(7*9+4*3+8*7))*48.7
20 END
```
b.
```
10 PRINT .278*(112+38+42)
20 END
```
c.
```
10 PRINT (88+78+84+49+63)/5
20 END
```

4:
```
10 PRINT ,"NAME"
20 PRINT
30 PRINT "LAST","FIRST","MIDDLE"
40 PRINT
50 PRINT "SMITH","JOHN","DAVID"
60 END
```

5:
```
10 PRINT ,"   BUDGET-APRIL"
20 PRINT "FOOD", 387.5
30 PRINT "RENT", 475.
40 PRINT "CAR",  123.71
50 PRINT "UTILITIES", 146.
60 PRINT "ENTERTAINMENT", 100.
70 PRINT , "------"
80 PRINT "TOTAL", 387.5+475.+123.71+146.+100.
90 END
```

6: a. 432
b. 76

2.3 Giving Names to Numbers and Words

In the examples and exercises of the preceding section, you probably noticed that you were wasting considerable time retyping certain numbers over and over. Not only is retyping a waste of time, it's also a likely source of errors. Fortunately, such retyping is unnecessary if you use variables.

A **variable** is a letter used to represent a number. Any letter of the alphabet may be used as a variable. (There are other possible names for variables. See below.) Possible variables are A,B,C,X,Y,or Z. When they are used as variables, lower case characters are considered different from their upper case equivalents; for example, B and b are different variables. At any given moment, a variable has a particular value. For example, the variable A might have the value 5 while B might have the value -2.137845. One method for changing the value of a variable is through use of the **LET** statement. The statement

```
10 LET X=7
```

sets the value of X equal to 7. Any previous value of X is erased. The **LET** command may be used to set the values of a number of variables simultaneously. For instance, the instruction

```
100 LET C=18: D=23: E=2.718
```

assigns C the value 18, D the value 23, and E the value 2.718.

Once you set the value of a variable, you may use the variable instead of the value throughout the program. For instance, if A has the value 7 then the expression

```
A+5
```

is evaluated as 7 + 5 or 12. The expression

```
3*A-10
```

is evaluated 3*7-10 = 21-10 = 11. The expression 2*A^2 is evaluated

```
2*7^2=2*49=98
```

TEST YOUR UNDERSTANDING 1 (answer on page 25)

Suppose that A has the value 4 and B has the value 3. What is the value of the expression A^2/2*B^2?

Note the following important fact:

If you do not specify a value for a variable, Floating Point BASIC will assign it the value 0.

Variables may also be used in **PRINT** statements. For example, the statement

```
10 PRINT A
```

will cause the computer to print the current value of A (in the first print zone, of course!). The statement

```
20 PRINT A,B,C
```

will result in printing the current values of A, B, and C in print zones 1, 2, and 3, respectively.

TEST YOUR UNDERSTANDING 2 (answer on page 25)

Suppose that A has the value 5. What will be the result of the instruction:

```
10 PRINT A,A^2,2*A^2
```

Example 1. Consider the three numbers 5.71, 3.23, 4.05. Calculate their sum, their product, and the sum of their squares (that is, the sum of their second powers; such a sum is often used in statistics).

Solution. Introduce the variables A,B, and C and set them equal, respectively, to the three numbers. Then compute the desired quantities.

```
10 LET A=5.71: B=3.23: C=4.05
20 PRINT "THE SUM IS", A+B+C
30 PRINT "THE PRODUCT IS", A*B*C
40 PRINT "THE SUM OF SQUARES IS", A^2+B^2+C^2
50 END
```

TEST YOUR UNDERSTANDING 3 (answer on page 25)

Consider the numbers 101,102,103,104,105,and 106. Write a program which calculates the product of the first two, the first three, the first four, the first five, and all six numbers.

The following mental imagery is often helpful in understanding how Floating Point BASIC handles variables. When Floating Point BASIC first encounters a variable, let's say A, it sets up a box (actually a memory location) which it labels"A". (See Figure 2-2.) In this box it stores the current value of A. When you request a change in the value of A, the computer throws out the current contents of the box and inserts the new value.

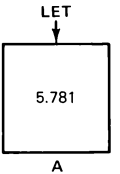


Figure 2-2. The variable A.

Note that the value of a variable need not remain the same throughout a program. At any point in the program, you may change the value of a variable (with a **LET** statement, for example). If a program is called on to evaluate an expression involving a variable, it will always use the current value of the variable, ignoring any previous values the variable may have had at earlier points in the program.

TEST YOUR UNDERSTANDING 4 (answer on page 25)

Suppose that a loan for $5,000 has an interest rate of 1.5 percent on the unpaid balance at the end of each month. Write a program to calculate the interest at the end of the first month. Suppose that at the end of the first month, you make a payment of $150 (after the interest is added). Design your program to calculate the balance after the payment. (Begin by letting B = the loan balance, I = the interest, and P = the payment. After the payment, the new balance is B + I - P.)

Example 2. What will be the output of the following computer program?

```
10 LET A= 10: B=20
20 LET A=5
30 PRINT A+B+C, A*B*C
40 END
```

Solution. Note that no value for C is specified, so C = 0. Also note that the value of A is initially set to 10. However, in line 20, this value is changed to 5. So in line 30, A, B, and C have the respective values 5, 20, and 0. Therefore, the output will be:

```
25              0
```

To the computer, the statement

```
LET A=
```

means that the current value of A is to be **replaced** with whatever appears to the right of the equal sign. Therefore, if you write

```
LET A=A+1
```

then you're asking the computer to replace the current value of A with A + 1. So if the current value of A is 4, the value of A after performing the instruction is 4 + 1, or 5.

TEST YOUR UNDERSTANDING 5 (answer on page 25)

What is the output of the following program?

```
10 LET A=5.3
20 LET A=A+1
30 LET A=2*A
40 LET A=A+8
50 PRINT A
60 END
```

Legal Variable Names

As mentioned earlier you may use any letter of the alphabet as a variable name. The Franklin is quite flexible concerning variable names. Any sequence of up to 238 characters which begins with a letter is a legal variable name. (For an exception, see below.) Therefore, you may use variables named PAYROLL, TAX, REFUND, and BALANCE. **However, Floating Point BASIC will use only the first two characters to distinguish one name from another.** Actually, not every sequence of characters is a legal variable name. You must avoid any sequences of characters which are reserved by Floating Point BASIC for special meanings. Examples of such words are:

IF,ON,OR,TO,PRINT,NEW,LIST,RUN,END,NEXT,DEL,FRE,GR, TEXT,DATA,USR

As a matter of fact, whenever you type any reserved word in lower case, your ACE will automatically display it in upper case. Once you become familiar with Floating Point BASIC, it'll be second nature to avoid these and the other reserved words as variable names.

Note: A variable name must always start with a letter.

A variable name **cannot** begin with a number. For example, 1A is **not** a legal variable name.

So far, all of the variables we have discussed have represented numerical values. However, Floating Point BASIC also allows variables to assume string constants as values. The variables for doing this are called **string variables.**

These are denoted by a variable name followed by a dollar sign. Thus, A$, B1$, and ZZ$ are all valid names of string variables. To assign a value to a string variable, use the **LET** statement with the desired value inserted in quotation marks after the equal sign. To set A$ equal to the string "BALANCE SHEET", we use the statement

```
LET A$="BALANCE SHEET"
```

We may print the value of a string variable just as we print the value of a numeric variable. For example, if A$ has the value just assigned, the statement

```
PRINT A$
```

will produce the following screen output:

```
BALANCE SHEET
```

Example 3. What will be the output of the following program?

```
10 LET A$="RECEIPTS":B$="EXPENSES"
20 LET A=20373.1: B=17584.31
30 PRINT A$,B$
40 PRINT A,B
50 END
```

Solution.

```
RECEIPTS                    EXPENSES
 20373.1                     17584.31
```

The variables A and A$ (as well as B and B$) are used in the same program. The variables A and A$ are considered **different** by the computer.

REMARKS IN PROGRAMS

It's very convenient to explain programs using remarks. For one thing, remarks make programs easier to read. Remarks also assist in finding errors and making modifications in a program. To insert a remark in a program, you may use the **REM** statement. For example, consider the line:

```
520 REM X DENOTES THE COST BASIS
```

Since the line starts with **REM**, the computer ignores it during program execution.

MULTIPLE STATEMENTS ON A SINGLE LINE

It's possible to put several Floating Point BASIC statements on a single line. Just separate them by a colon. For example, instead of the two statements:

```
10 LET A=5.784: B=3.571
20 PRINT A^2+B^2
```

use the single statement:

```
10 LET A=5.784: B=3.571 : PRINT A^2+B^2
```

To insert a remark on the same line as a program statement, use a colon followed by a **REM**, as in this example:

```
10 LET A=PI*R^2 :REM A IS THE AREA,R IS THE RADIUS
```

The computer ignores the remainder of the line after the **REM** statement. In what follows, we will sprinkle comments liberally throughout your programs so that they will be easier to understand.

TEST YOUR UNDERSTANDING 6 (answer on page 25)

What is the result of the following lines?

```
10 LET A=7:B$="COST":C$="TOTAL"
20 PRINT C$,B$
30 PRINT "=",A
```

Exercises (answers on page 196)

In Exercises 1-6, determine the output of the given program.

1.
```
10 LET A=5:B=5
20 PRINT A+B
30 END
```

2.
```
10 LET AA=5
20 PRINT AA*B
30 END
```

3.
```
10 LET A1=5
20 PRINT A1^2+5*A1
30 END
```

4.
```
10 LET A=2: B=7:C=9
20 PRINT A+B: A-C,A*C
30 END
```

5.
```
10 LET A$="JOHN JONES"
20 LET B$="AGE": C=38
30 PRINT A$, B$, C
40 END
```

6.
```
10 LET X=11, Y=19
20 PRINT 2*X
30 PRINT 3*Y
40 END
```

What's wrong with the following Floating Point BASIC statements?

7. `10 LET A="YOUTH"`
8. `10 LET AA=-12`
9. `10 LET A$=57`
10. `LET ZZ$=Address`
11. `250 LET AAA=-9`
12. `10000 LET 1A=-2.34567`
13. Consider the numbers 2.3758, 4.58321, and 58.11. Write a program which computes their sum, product, and the sum of their squares.

14. A company has three divisions: Office Supplies, Computers, and Newsletters. The revenues of these three divisions for the preceding quarter were, respectively, $346,712, $459,321, and $376,872. The expenses for the quarter were $176,894, $584,837, and $402,195 respectively. Write a program which displays this data on the screen, with appropriate explanatory headings. Your program should also compute and display the net profit (loss) from each division and the net profit (loss) for the company as a whole.

ANSWERS TO TEST YOUR UNDERSTANDING 1,2,3,4,5, and 6

1: .89

2: It prints the display:

```
 5          25          50
```

3:
```
10 LET A=101:B=102:C=103:D=104:E=105:F=106
20 PRINT A*B
30 PRINT A*B*C
40 PRINT A*B*C*D
50 PRINT A*B*C*D*E
60 PRINT A*B*C*D*E*F
70 END
```

4:
```
10 LET B=5000: I=.015: P=150.
20 IN=I*B
30 PRINT "INTEREST EQUALS", IN
40 B=B+IN
50 PRINT " BALANCE WITH INTEREST EQUALS", B
60 B=B-P
70 PRINT "BALANCE AFTER PAYMENT EQUALS", B
80 END
```

5: 12.6

6: It creates the display:

```
TOTAL        COST
=            7
```

2.4 Doing Repetitive Operations

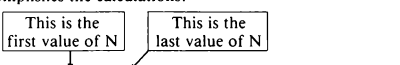
Suppose that you wish to solve 50 similar multiplication problems. It's certainly possible to type in the 50 problems one at a time and let the computer solve them. However, this is a very clumsy way to proceed. Suppose that instead of 50 problems there were 500, or even 5000. Typing the problems one at a time would not be practical. If, however, you can describe to the computer

the entire class of problems you want solved, then you can instruct the computer to solve them using only a few Floating Point BASIC statements.

Let's consider a concrete problem. Suppose that you wish to calculate the quantities

$1^2, 2^2, 3^2, \ldots, 10^2$

That is, you wish to calculate a table of squares of integers from 1 to 10. This calculation can be described to the computer as calculating N 2, where the variable N is allowed to assume, one at a time, each of the values 1,2,3,...,10. Here's a sequence of Floating Point BASIC statements which accomplishes the calculations:




```
10 FOR N=1 TO 10
20 PRINT N*N
30 NEXT N
40 END
```

lines 10-20-30 repeated 10 times

The sequence of statements in lines 10,20, and 30 is called a **loop.** When the computer encounters the **FOR** statement, it sets N equal to 1 and continues executing the statements. Line 20 calls for printing N^2. Since N is equal to 1, you have N^2 = 1^2 = 1. So the computer will print a 1. Next comes statement 30, which calls for the next N. This instructs the computer to return to the **FOR** statement in 10, increase N to 2, and to repeat instructions 20 and 30. This time, N^2 = 2^2 = 4. Line 20 then prints a 4. Line 30 says to go back to line 10 and increase N to 3 and so forth. Lines 10, 20, and 30 are repeated 10 times! After the computer executes lines 10,20,and 30 with N = 10, it leaves the loop and executes line 40.

Type in the above program and give the **RUN** command. The output will look like this:

```
1
4
9
16
25
36
49
64
81
100
]
```

TEST YOUR UNDERSTANDING 1 (answers on page 34)

a. Devise a loop allowing N to assume the values 3 to 77.
b. Write a program which calculates N^2 for N = 3 to 77.

Let's modify the above program to include on each line of output not only N^2, but also the value of N. To make the table easier to read, let's also add two column headings. The new program reads:

```
10 PRINT "N",,"N^2"
20 FOR N=1 TO 10
30 PRINT N,,N*N
40 NEXT N
50 END
```

The output now looks like this:

```
N               N^2
1               1
2               4
3               9
4               16
5               25
6               36
7               49
8               64
9               81
10              100
] ■
```

TEST YOUR UNDERSTANDING 2 (answer on page 34)

What would happen if you changed the number of line 10 to 25?

Let's now illustrate some of the many uses loops have by means of some examples.

Example 1. Write a Floating Point BASIC program to calculate $1+2+3+\ldots+100$.

Solution. Let's use a variable S (for sum) to contain the sum. Let's start S at 0 and use a loop to successively add to S the numbers 1,2,3,...,100. Here's the program.

```
10 LET S=0
20 FOR N=1 TO 100
30 LET S=S+N
40 NEXT N
50 PRINT S
60 END
```

These instructions are repeated 100 times

When you enter the loop the first time, S = 0 and N = 1. Line 30 then replaces S by S + N, or 0 + 1. Line 40 sends you back to line 20, where the value of N is now set equal to 2. In line 30, S (which is now 0 + 1) is replaced by S + N, or 0 + 1 + 2. Line 40 now sends you back to line 20, where N is now set equal to 3. Line 30 then sets S equal to 0 + 1 + 2 + 3. Finally, on the 100th time through the loop, S is replaced by 0 + 1 + 2 + ... + 100, the desired sum. If you run the program, you derive the output

```
5050
```

TEST YOUR UNDERSTANDING 3 (answer on page 34)

Write a Floating Point BASIC program to calculate 101 + 102 + ... + 110.

TEST YOUR UNDERSTANDING 4 (answer on page 34)

Write a Floating Point BASIC program to calculate and display the numbers 2,2^2,2^3,...,2^20.

Example 2. Write a program to calculate the sum:

```
1x2+2x3+3x4+...+ 49x50
```

Solution. Let the sum be contained in the variable S, like you did in Example 1. The quantities to be added are just the numbers N*(N + 1) for N = 1,2,3,...,49. So here's the program:

```
10 LET S=0
20 FOR N=1 TO 49
30 LET S=S+N*(N+1)
40 NEXT N
50 PRINT S
60 END
```

Example 3. You borrow $7,000 to buy a car. You finance the balance for 36 months at an interest rate of one percent per month. Your monthly payments

are $232.50. Write a program which computes the amount of interest each month, the amount of the loan which is repaid, and the balance owed.

Solution. Let B denote the balance owed. Initially you have B equal to 7,000 dollars. At the end of each month compute the interest (I) owed for that month, namely .01*B. For example, at the end of the first month, the interest owed is .01*7000.00 = $70.00. Let P = 232.50 to denote the monthly payment, and let R denote the amount repaid out of the current payment. Then R = P–I. For example, at the end of the first month, the amount of the loan repaid is 232.50–70.00 = 162.50. The balance owed may then be calculated as B–R. At the end of the first month, the balance owed is 7000.00–162.50 = 6837.50. Here's a program which performs these calculations:

```
10 PRINT "INTEREST","BALANCE"
20 LET B=7000
25 LET P=232.5
30 FOR M=1 TO 36:REM M IS MONTH NUMBER
40 LET I=.01*B :REM  CALCULATE INTEREST FOR MONTH
50 LET R=P-I :REM  CALCULATE REPAYMENT
60 LET B=B-R :REM   CALCULATE NEW BALANCE
70 PRINT I,B
80 NEXT M
90 END
```

You should attempt to run this program. Notice that it runs, but it's pretty useless because the screen won't contain all of the output. Most of the output goes flying by before you can read it. One method for remedying this situation is to press [BREAK] as the output scrolls by on the screen. This will stop execution of the program and freeze the contents of the screen. To resume execution and unfreeze the screen, type **CONT** and hit the [RETURN] key. The output will begin to scroll again. To use this technique requires some manual dexterity. Moreover, it's not possible to guarantee where the scrolling will stop.

TEST YOUR UNDERSTANDING 5

RUN the program of Example 3 and practice freezing the output on the screen. It may take several runs before you are comfortable with the procedure.

There's another way to solve the problem. The [PAUSE] key will make a program "pause" in whatever it's doing, in most cases. Hitting [PAUSE] will stop the display on the screen indefinitely. Typing [PAUSE] again will restart the program.

Let's now describe another method of adapting the output to your screen size by printing only 12 months of data at one time. This amount of data will fit since the screen contains 24 lines. Use a second loop to keep track of 12

month periods. The variable for the new loop will be Y (for "years"), and Y will go from 0 to 2. The month variable will be M as before, but now M will go only from 1 to 12. The month number will now be 12*Y + M (the number of years plus the number of months). Here's the revised program.

```
10 LET B=7000
15 LET P=232.5
20 FOR Y=0 TO 2
30 PRINT "INTEREST","BALANCE"
40 FOR M=1 TO 12
50 LET I=.01*B
60 LET R=P-I: REM ONE 12 MONTH PERIOD
70 LET B=B-R
80 PRINT I,B
90 NEXT M
100 STOP :REM HALTS EXECUTION
110 HOME :REM CLEAR SCREEN
120 NEXT Y :REM GOES TO NEXT 12 MONTHS
130 END
```

This program utilizes several new statements. In line 100, the **STOP** statement causes the computer to stop execution of the program. The computer remembers where it stops, however, and all values of the variables are preserved. The **STOP** statement also leaves unchanged the contents of the screen. You can take as long as you wish to examine the data on the screen. When you're ready for the program to continue, type **CONT.** The computer will resume where it left off. The first instruction it encounters is in line 110. **HOME** clears the screen. So, after being told to continue, the computer clears the screen and goes on to the next value of Y—the next 12 months of data. Here's a copy of the output. The underlined statements are those you type.

```
] ■

RUN

INTEREST            BALANCE
70                  6837.5
68.375              6673.375
66.73375            6507.6.0875
65.0760875          6340.18484
63.4018484          6171.08669
61.7108669          6000.29755
60.0029755          5827.80053
58.2780053          5653.57854
56.5357854          5477.61432
54.7761432          5299.89047
52.9989047          5120.38937
51.2038937          4939.09326
```

```
BREAK IN 100

]■
]■ CONT


INTEREST            BALANCE
49.3909326          4755.98419
47.5598419          4571.04404
45.7104404          4384.25448
43.8425448          4195.59702
41.9559702          4005.05299
40.0505299          3812.60352
38.1260352          3618.22955
36.1822955          3421.91185
34.2191185          3223.63097
32.2363097          3023.36728
30.2336728          2821.10095
28.2110095          2616.81196

BREAK IN 100

]■

]■ CONT


INTEREST            BALANCE
26.1681196          2410.48008
24.1048008          2202.08488
22.0208488          1991.60573
19.9160573          1779.02179
17.7902179          1564.31201
15.6431201          1347.45513
13.4745513          1128.42968
11.2842968          907.213974
9.07213974          683.7856113
6.83786113          458.123975
4.58126             230.205214
2.30205215          7.266581066E-03
BREAK IN 100

]■
```

Note that the data in the output is carried out to ten figures, even though the problem deals with dollars and cents. You'll look at the problem of rounding numbers later. Also note the balance listed at the end of month 36. It's in scientific notation. The -03 indicates that the decimal point is to be moved three places to the left. The number listed is .00726658106 or about .72 cents

(less than one cent)! The computer shifted to scientific notation since the usual notation (.007034302) requires more than ten digits. The computer made the choice of which form of the number to display.

Using Loops to Create Delays

By using a loop we can create a delay inside the computer. Consider the following sequence of instructions:

```
10 FOR N=1 TO 3000
20 NEXT N
```

This loop doesn't do anything! However, the computer repeats instructions 10 and 20 three thousand times! This may seem like a lot of work. But not for a computer. To obtain a feel for the speed at which the computer works, you should time this sequence of instructions. Such a loop may be used as a delay. For example, when you wish to keep some data on the screen without stopping the program, just build in a delay. Here's a program which prints two screens of text. A delay is imposed to give a person time to read the first screen.

```
10 PRINT "THIS IS A PROGRAM TO DISPLAY SALES"
20 PRINT "FOR THE YEAR TO DATE"
30 FOR N=1 TO 5000
40 NEXT N: REM DELAY LOOP
50 HOME
60 PRINT "YOU MUST SUPPLY THE PARAMETERS:"
70 PRINT "PRODUCT, TERRITORY, VOLUME"
80 END
```

Example 4. Use a loop to produce a blinking display for a security system.

Solution. Suppose that your security system is tied in with your computer and the system detects that an intruder is in your warehouse.. Let's print out the message:

```
SECURITY SYSTEM DETECTS INTRUDER-ZONE 2
```

For attention, let's blink this message on and off by alternately printing the message and clearing the screen.

```
10 FOR N=1 TO 2000
20 PRINT:PRINT:PRINT
30 PRINT "SECURITY SYSTEM DETECTS INTRUDER-ZONE 2"
40 FOR K=1 TO 150
50 NEXT K
60 HOME
70 FOR K=1 TO 150
80 NEXT K
90 NEXT N
100 END
```

The loop in 30-40 is a delay loop to keep the message on the screen a moment. Line 60 turns the message off, but the **PRINT** statement in line 30 almost immediately turns it back on. The message will blink 2000 times.

TEST YOUR UNDERSTANDING 6 (answer on page 34)

Write a program which blinks your name on the screen 500 times, leaving your name on the screen for a loop of length 50 each time.

In all of the loop examples, the loop variable increased by one with each repetition of the loop. However, it's possible to have the loop variable change by any amount. For example, the instructions

```
10 FOR N=1 TO 5000 STEP 2
.
.
.
1000 NEXT N
```

define a loop in which N jumps by 2 for each repetition, so N will assume the values:

```
1,3,5,7,9,...,4999
```

Similarly, use of `STEP .5` in the above loop will cause N to advance by .5 and assume the values:

```
1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, ..., 5000
```

It's even possible to have a negative step. In this case, the loop variable will run backwards. For example, the instructions

```
10 FOR N=100 TO 1 STEP -1
.
.
.
100 NEXT N
```

will "count down" from N = 100 to N = 1 one unit at a time. You'll get some applications of such instructions in the exercises.

TEST YOUR UNDERSTANDING 7 (answers on page 34)

Write instructions to allow N to assume the following sequences of values:

a. 95,96.7,98.4,...,112
b. 200,199.5,199,...,100

Exercises (answers on page 197)

Write Floating Point BASIC programs to compute the following quantities.

1. $1^2 + 2^2 + 3^2 + \ldots + 25^2$
2. $1 + (1/2) + (1/2)^2 + \ldots + (1/2)^{10}$
3. $1^3 + 2^3 + 3^3 + \ldots + 10^3$
4. $1 + (1/2) + (1/3) + \ldots + (1/100)$
5. Write a program to compute N^2, N^3 for N = 1,...,12. The format of your output should be as follows:

```
N          N^2        N^3
1
2
3
.
.
.
12
```

6. Suppose that you have a car loan whose current balance is $4,000.00. The monthly payment is $125.33 and the interest is one percent per month on the unpaid balance. Make a table of the interest payments and balances for the next 12 months.
7. Suppose you deposit $1,000 on January 1 of each year into a savings account paying 10 percent interest. Suppose that the interest is computed on January 1 of each year, based on the balance for the preceding year. Calculate the balances in the account for each of the next 15 years.
8. A stock market analyst predicts that Tyro Computers, Inc. will achieve a 20 percent growth in sales in each of the next three years, but profits will grow at a 30 percent annual rate. Last year's sales were $35 million and last year's profits were $5.54 million. Project the sales and profits for the next three years, based on the analyst's prediction.

ANSWERS TO TEST YOUR UNDERSTANDING 1,2,3,4,6 and 7

1: a.
```
10 FOR N=3 TO 77
   .
   .
100 NEXT N
```

b.
```
10 FOR N=3 TO 77
   20 PRINT N^2
   30 NEXT N
   40 END
```

2: The heading

```
N       N^2
```

would be printed before each entry of the table.

3:
```
10 S=0
20 FOR N=101 TO 110
30 S=S+N
```

```
    40 NEXT N
    50 PRINT S
    60 END
4:  10 FOR N=1 TO 20
    20 PRINT 2^N
    30 NEXT N
    40 END
6:  10 FOR N-1 TO 500
    20 PRINT "<YOUR NAME>"
    30 FOR K=1 TO 50
    40 NEXT K
    50 HOME
    60 NEXT N
    70 END
7:  a.  10 FOR N=95 TO 112 STEP 1.7
    b.  20 FOR N=200 TO 100 STEP -.5
```

2.5 Some Floating Point BASIC Commands

Thus far, most of our attention has been focused on learning statements to insert inside programs. Let's now learn a few of the commands available for manipulating programs and the computer. The **NEW** command, previously discussed, is in this category. Remember the following facts about Floating Point BASIC commands:

1. Commands are typed **without** using a line number.
2. You must hit the [RETURN] key after typing a command.
3. A command may be given whenever the computer is in command mode. (Recall that when the computer first enters the command mode, it displays the] ■ message. The computer remains in the command mode until a **RUN** command is given.)
4. The computer executes commands as soon as they're received.

Listing a Program

To obtain a list of all the statements of a current program in RAM, you may type the command:

```
LIST
```

For example, suppose that RAM contains the following program.

```
10 PRINT 5+7, 5-7
20 PRINT 5*7,5/7
30 END
```

(This program may or may not be currently displayed on the screen.) If you type **LIST**, then the above three instruction lines will be displayed, followed by the prompt] ■.

In developing a program, you'll undoubtedly find it necessary to input lines in non-consecutive order and to correct lines already input. If this happens, the screen will usually not indicate the current version of the program. Typing **LIST** every so often will assist in keeping track of what has been changed. LISTing is particularly helpful in checking a program or determining why a program won't run.

Note that the Franklin screen can display up to 24 lines of text. This means you can display, at most, 24 program statements at one time. Therefore, you must list long programs a section at a time. For example, to LIST only those statements with line numbers from 100 to 240, use the command:

```
LIST 100-240
```

or

```
LIST 100,240
```

In a similar fashion, you may list any collection of consecutive program lines.

There are several other variations of the **LIST** command. To list the program lines from the beginning of the program to line 75, use the command:

```
LIST -75
```

Similarly, to list the program lines from 100 to the end of the program, use the command:

```
LIST 100-
```

To list line 100, use the command:

```
LIST 100
```

TEST YOUR UNDERSTANDING 1 (answers on page 37)

Write a command to:

a. List line 200
b. List lines 300-330
c. List lines 300 to the end

Test out these commands with a program.

Deleting Program Lines

When typing a program or revising an existing program, it's often necessary to delete lines which are already part of the program. One simple way is to type the line number followed by RETURN. For example,

```
275
```

(followed by hitting the RETURN key) will delete line 275. The **DEL** command may also be used for the same purpose. For example, you may delete line 275, using the command

```
DEL 275,275
```

Similarly, to delete the lines 500 to 700 inclusive, use the command:

```
DEL 500,700
```

If the program does not have a line 700, the computer will determine the last line before 700 and will delete from 500 to that line, inclusive.

TEST YOUR UNDERSTANDING 2 (answers on page 37)

What is wrong with the following commands?

a. DEL 450,
b. LIST 450-
c. DEL 300,200

ANSWERS TO TEST YOUR UNDERSTANDING 1 and 2

1: a. `LIST 200`
   b. `LIST 300-330`
   c. `LIST 300-`

2: a. The line number of the last line to be deleted must be specified. It should read:
   ```
   DEL 405,450
   ```
   b. Nothing wrong.
   c. The lower line number must come first. The command should read:
   ```
   DEL 200,300
   ```

2.6 Letting Your Computer Make Decisions

One of the principal features which makes computers useful as problem-solving tools is their ability to make decisions. The vehicle which Floating Point BASIC uses to make decisions is the **IF . . . THEN** statement. The **IF** part of such a statement allows you to ask a question. If the answer is YES, then the computer carries out the **THEN** part of the statement. If the answer is NO, then the computer goes on to the next statement in numerical order. For example, consider the statement

```
500 IF N=0 THEN PRINT "CALCULATION DONE"
```

First, the computer determines if N is equal to zero. If so, it prints "CALCULATION DONE" and proceeds with the next instruction after line 500. However, if N is not zero, then the computer goes directly to the next instruction after line 500 and continues program execution from that instruction.

A variation of the **IF . . . THEN** statement allows you to insert a line number after **THEN.** For example, the instruction

```
600 IF N > 0 THEN GOTO 500
```

will determine whether N is greater than 0. If so, the program will go to line 500. Otherwise, the program will go to the next line in numerical sequence. The **GOTO** after **THEN** may actually be omitted, but putting the **GOTO** in will never hurt anything.

After **IF,** you may insert any expression which the computer may test for truth or falsity. Here are some examples:

```
N=0
N > 5   (N is greater than 5)
N < 12.9   (N is less than 12.9)
N >= 0   (N is greater than or equal to 0)
N <= -1   (N is less than or equal to -1)
N >< 0   (N is not equal to 0)
A+B <> C (A + B is not equal to C)
A^2+B^2 <= C^2 (A^2 + B^2 is less than or equal to C^2)
```

TEST YOUR UNDERSTANDING 1 (answers on page 48)

Write instructions which do the following:

a. If A is less than B, then print the value of A plus B; if not, then go to the end.
b. If A^2 + D is at least 5000, then go to line 300; if not, go to line 500.
c. If N is larger than the sum of I and K, then set N equal to the sum of I and K; otherwise, let N equal K.

The **IF ... THEN** statement may be used to interrupt the normal sequence of program line execution, based upon the truth or falsity of some condition. In many applications, however, you'll want to perform instructions out of the normal sequence, independent of any conditions. For such applications, you can use the **GOTO** instruction. This instruction has the form:

```
GOTO < line number >
```

For example, the instruction

```
1000 GOTO 300
```

will send the computer back to line 300 for its next instruction.

NOTE: Any statement that follows on the same line as an **IF . . . THEN** statement will be executed only if the **THEN** part is executed. For example, in the statement

```
10 IF A < B THEN C=D: GOTO 300
20 C=E
```

the computer will do the following: If A is less than B, the computer will set C equal to D. It will then go to 300. On the other hand, if A is not less than B, the computer will go to the statement in line 20.

The following examples illustrate some of the uses of the **IF ... THEN** and **GOTO** statements.

Example 1. A lumber supply house has a policy that a credit invoice may not exceed $1,000, including a 10 percent processing fee and 5 percent sales tax. A customer orders 150 2x4 studs at $1.99 each, 30 sheets of plywood at $14.00 each, 300 pounds of nails at $1.14 per pound, and two double hung insulated windows at $187.95 each. Write a program which prepares an invoice and decides whether the order is over the credit limit.

Solution. Let's use the variables A1,A2,A3,and A4 to denote, respectively, the numbers of studs, sheets of plywood, pounds of nails, and windows. Let's use the variables B1,B2,B3,and B4 to denote the unit costs of these four items. The cost of the order is then computed as:

```
A1*B1+A2*B2+A3*B3+A4*B4
```

Add 10 percent of this amount to cover processing and form the sum to obtain the total order. Compute 5 percent of the last amount as tax and add it to the total to obtain the total amount due. Finally, determine if the total amount due is more than $1,000. If it is, print out the message: `ORDER EXCEEDS $1,000. CREDIT SALE NOT PERMITTED.` Here's your program.

```
10 LET A1=150:A2=30:A3=300:A4=2
20 LET B1=1.99:B2=14.:B3=1.14:B4=187.95
30 LET T= A1*B1+A2*B2+A3*B3+A4*B4
40 PRINT "TOTAL ORDER",T
50 LET P=.1*T
```

```
60 PRINT "PROCESSING FEE",P
70 LET TX=.05*(P+T)
80 PRINT "SALES TAX",TX
90 DU=T+P+TX
100 PRINT "AMOUNT DUE", DU
110 IF DU > 1000 THEN GOTO 200
120 GOTO 300
200 PRINT "ORDER EXCEEDS $1,000"
210 PRINT "CREDIT SALE NOT PERMITTED"
220 GOTO 400
300 PRINT "CREDIT SALE OK"
400 END
```

Note the decision in line 110. If the amount due exceeds $1,000, then the computer goes to line 200 where it prints out a message denying credit. In line 220, the computer is sent to line 400 which is the **END** of the program. On the other hand, if the amount due is less than $1,000, the computer is sent to line 300, in which credit is approved.

TEST YOUR UNDERSTANDING 2 (answers on page 48)

Suppose that a credit card charges 1.5 percent per month on any unpaid balance up to $500 and 1 percent per month on any excess over $500.

a. Write a program which computes the service charge and the new balance.
b. Test your program on the unpaid balances of $1,300 and $275.

TEST YOUR UNDERSTANDING 3 (answers on page 48)

Consider the following sequence of instructions.

```
100 IF A>=5 THEN GOTO 200
110 IF A>=4 THEN GOTO 300
120 IF A>=3 THEN GOTO 400
130 IF A>=2 THEN GOTO 500
```

Suppose that the current value of A is 3. List the sequence of line numbers which will be executed.

Example 2. At $20 per square yard, a family can afford up to 500 square feet of carpet for their dining room. They wish to install the carpet in a circular shape. It's been decided that the radius of the carpet is to be a whole number of feet. What is the radius of the largest carpet they can afford? (The area of a circle of radius "R" is PI times R^2, where PI equals approximately 3.14159.)

Solution. Let's compute the area of the circle of radius 1,2,3,4,... and determine which of the areas are less than 500.

```
10 LET PI=3.14159
20 LET R=1 : REM R IS THE RADIUS OF THE CIRCLE
30 LET A=PI*R^2 : REM A IS THE AREA OF THE CIRCLE
40 IF A >=500 THEN GOTO 100 : REM IF AREA IS AT LEAST
   500, END
50 PRINT R : REM IF AREA IS LESS THAN 500, PRINT R
60 LET R=R+1 : REM GO TO NEXT RADIUS
70 GOTO 30
100 END
```

Note that line 40 contains an **IF ... THEN** statement. If A, as computed in line 30, is 500 or more, then the computer goes to line 100, the **END.** If A is less than 500, the computer proceeds to the next line, namely 50. It then prints out the current radius, increases the radius by 1, and goes back to line 30 to repeat the entire procedure. Note that lines 30-40-50-60-70 are repeated until the area becomes at least 500. In effect, this sequence of five instructions forms a loop. However, a **FOR ... NEXT** instruction wasn't used because you couldn't know in advance how many times you wanted to execute the loop. The computer decided the stopping point via the **IF ... THEN** instruction.

Example 3. A school board race involves two candidates. The returns from the four wards of the town are as follows:

	Ward 1	Ward 2	Ward 3	Ward 4
Mr. Thompson	487	229	1540	1211
Ms. Wilson	1870	438	110	597

Calculate the total number of votes achieved by each candidate, determine the percentage achieved by each candidate, and decide who won the election.

Solution. Let A1,A2,A3,and A4 be the totals for Mr. Thompson in the four wards; let B1–B4 be the corresponding numbers for Ms. Wilson. Let TA and TB denote the total votes, respectively, for Mr. Thompson and Ms. Wilson. Here's the program:

```
10 LET A1=487: A2=229: A3=1540: A4=1211
20 LET B1=1870: B2=438: B3=110: B4=597
30 LET TA=A1+A2+A3+A4 : REM TOTAL FOR THOMPSON
40 LET TB=B1+B2+B3+B4 : REM TOTAL FOR WILSON
50 LET T=TA+TB : REM TOTAL VOTES CAST
60 LET PA=100*TA/T : REM PERCENTAGE FOR THOMPSON
65 REM TA/T IS THE FRACTION OF VOTES FOR THOMPSON
66 REM MULTIPLY BY 100 TO CONVERT TO A PERCENTAGE
70 LET PB=100*TB/T : REM PERCENTAGE FOR WILSON
110 LET A$="THOMPSON"
120 LET B$="WILSON"
```

```
130 REM 140-170 PRINT THE PERCENTAGES OF THE CANDIDATES
140 PRINT "CANDIDATE", "VOTES","PCT"
150 PRINT A$,TA, PA
160 PRINT B$,TB, PB
170 REM 180-400 DECIDE THE WINNER
180 IF TA > TB THEN GOTO 300
190 IF TA < TB THEN GOTO 400
200 PRINT "TIE VOTE!"
210 GOTO 1000
300 PRINT A$, "WINS"
305 PRINT " "
310 GOTO 1000
400 PRINT B$, "WINS"
410 PRINT " "
1000 END
```

Note the logic used for deciding who won. In line 180 you compare the votes TA and TB. If TA is the larger, then A (Thompson) is the winner. Then go to 300, print the result, and **END**. On the other hand, if TA > TB is **false**, then either B wins or the two are tied. According to the program, if TA > TB is false, you go to line 190, where you determine if TA < TB. If this is true, then B is the winner; you go to 400, print the result, and **END**. On the other hand, if TA < TB is false, then the only possibility left is that TA = TB. According to the program, if TA = TB you go to 200, where you print the proper result, and then **END**.

Infinite Loops and the CTRL C Key

As we have seen above, it's very convenient to be able to execute a loop without knowing in advance how many times the loop will be executed. However, with this convenience comes a danger. It's perfectly possible to create a loop which will be repeated an infinite number of times! For example, consider the following program:

```
10 LET N=1
20 PRINT N
30 LET N=N+1
40 GOTO 20
50 END
```

The variable N starts off at 1. You print it and then increase N by 1 (to 2), print it, increase N by 1 (to 3), print it, and so forth. This program will go on forever! Such programs should clearly be avoided. However, even experienced programmers occasionally create infinite loops. When this happens, there is no need to panic. There is a way of stopping the computer. Just press BREAK (or the keys CTRL and C simultaneously). This will interrupt the program currently in progress and return the computer to command mode.

The computer is then ready to accept a command from the keyboard. Note, however, that any program in RAM is undisturbed.

TEST YOUR UNDERSTANDING 4

Type the above program, **RUN** it and stop it using the CTRL C combination. After stopping it, **RUN** the program again.

The INPUT Statement

It's very convenient to have the computer request information from you while the program is actually running. This can be accomplished via the **INPUT** statement. To see how, consider the statement

```
570 INPUT A
```

When the computer encounters this statement in the course of executing the program, it types out a ? and waits for you to respond by typing the desired value of A (and then hitting the RETURN key). The computer then sets A equal to the numeric value you specified and continues running the program.

You may use an **INPUT** statement to specify the values of several different variables at one time. These variables may be numeric or string variables. For example, suppose that the computer encounters the statement:

```
50 INPUT A,B,C$
```

It will respond with

```
?
```

You then type in the desired values for A,B, and C$, in the same order as in the program, and separated by commas. For example, suppose that you type

```
10.5, 11.42, BEARINGS
```

followed by a RETURN. The computer will then set:

```
A=10.5, B=11.42, C$="BEARINGS"
```

If you respond to the above question mark by typing only a single number, 10.5, for example, the computer will respond with

```
??
```

to indicate that it expects more data. If you attempt to specify a string constant where you should have a numeric constant, the computer will respond with the message

```
? REENTER
```

and will wait for you to repeat the **INPUT** operation.

It's helpful to include a prompting message which describes the input the computer is expecting. To do so, just put the message in quotation marks after the word **INPUT** and place a semicolon after the message (before the list of variables to be input). For example, consider the statement

```
175 INPUT "ENTER COMPANY, AMOUNT?"; A$, B
```

When the computer encounters this program line, the dialog will be as follows:

```
ENTER COMPANY, AMOUNT? AJAX OFFICE SUPPLIES, 2579.48
```

The underlined portion indicates your response to the prompt. The computer will now assign the values:

```
A$= "AJAX OFFICE SUPPLIES", B=2579.48
```

TEST YOUR UNDERSTANDING 5 (answer on page 48)

Write a program which allows you to set variables A and B to any desired values via an **INPUT** statement. Use the program to set A equal to 12 and B equal to 17.

The next two examples illustrate the use of the **INPUT** statement, and provide further practice in using the **IF ... THEN** statement.

Example 4. You're a teacher compiling semester grades. Suppose there are four grades for each student and that each grade is based on the traditional 0 to 100 scale. Write a program which accepts the grades as input, computes the semester average, and assigns grades according to the following scale:

```
90-100      A
80-89.9     B
70-79.9     C
60-69.9     D
< 60        F
```

Solution. Use an **INPUT** statement to enter the grades into the computer. Your program will allow you to compute the grades of students, one after the other, via a loop. You may terminate the loop by entering a negative grade. Here's the program.

```
5 HOME
10 PRINT "ENTER STUDENT'S 4 GRADES."
20 PRINT "SEPARATE GRADES BY COMMAS."
30 PRINT "FOLLOW LAST GRADE WITH RETURN."
40 PRINT "TO END PROGRAM, ENTER NEGATIVE GRADE."
50 INPUT A1,A2,A3,A4
```

```
60 IF A1 < 0 THEN END
70 IF A2 < 0 THEN END
80 IF A3 < 0 THEN END
90 IF A4 < 0 THEN END
100 LET A=(A1+A2+A3+A4)/4
110 PRINT "SEMESTER AVERAGE"; A
120 IF A >=90 THEN PRINT "SEMESTER GRADE=A" : GOTO 10
130 IF A >=80 THEN PRINT "SEMESTER GRADE=B" : GOTO 10
140 IF A >=70 THEN PRINT "SEMESTER GRADE=C" : GOTO 10
150 IF A >=60 THEN PRINT "SEMESTER GRADE=D" : GOTO 10
160 PRINT "SEMESTER GRADE = F":GOTO 10
200 END
```

Note the logic for printing out the semester grades. First compute the semester average A. In line 120 you ask if A is greater than or equal to 90. If so, you assign the grade A, and go to the **END.** In case A is less than 90, line 120 sends the computer to line 130. In line 130, you ask if A is greater than or equal to 80. If so, then you assign the grade B. (The point is that the only way you can get to line 130 is for A to be less than 90. So if A is greater than or equal to 80, you know that A lies in the B range.) If not, you go to line 140, and so forth. This logic may seem a trifle confusing at first, but after repeated use, it'll seem quite natural.

Example 5. Write a program to maintain your checkbook. The program should allow you to record an initial balance, enter deposits, and enter checks. It should also warn you of overdrafts.

Solution. Let the variable B always contain the current balance in the checkbook. The program will ask for the type of transaction you wish to record. A "D" will mean that you wish to record a deposit; a "C" will mean that you wish to record a check; a "Q" will mean that you're done entering transactions and wish to terminate the program. After entering each transaction, the computer will figure your new balance, report it to you, check for an overdraft, and report any overdraft to you. In case of an overdraft, the program will allow you to cancel the preceding check!

```
10 INPUT "WHAT IS YOUR STARTING BALANCE:"; B
20 INPUT "WHAT TRANSACTION TYPE (D,C,OR Q):"; A$
30 IF A$="Q" THEN END
40 IF A$="D" THEN INPUT "DEPOSIT AMOUNT"; D:GOTO 110
100 IF A$ ="C" THEN 200
110 LET B=B+D : REM ADD DEPOSIT TO BALANCE
120 PRINT "YOUR NEW BALANCE IS"; B
130 GOTO 20
200 INPUT "CHECK AMOUNT"; C
210 LET B=B-C : REM DEDUCT CHECK AMOUNT
220 IF B < 0 THEN GOTO 300 : REM TEST FOR OVERDRAFT
230 PRINT "YOUR NEW BALANCE IS"; B
```

ANSWERS TO TEST YOUR UNDERSTANDING 1,2,3, and 5

```
1:  a.  10 IF A<B THEN PRINT A+B : END
        20 END
    b.  10 IF A^2+B >= 5000 THEN GOTO 300
        20 GOTO 500
    c.  10 IF N>I+K THEN N=I+K
        20 N=K
2:  10 INPUT "UNPAID BALANCE";B
    20 IF B > 500 THEN GOTO 100 : GOTO 200
    30 GOTO 200
    100 LET C=B-500
    110 IN=.015*500+.01*C
    120 GOTO 300
    200 IN=.015*B
    300 PRINT "INTEREST EQUALS";IN
    310 PRINT "NEW BALANCE EQUALS";B+IN
    320 END
3:  100-110-120-400
5:  10 INPUT "THE VALUES OF A AND B ARE";A,B
    20 END
```

2.7 Some Programming Tips

Now that you've learned the most elementary Floating Point BASIC commands and statements, let's discuss a few topics which will make programming easier.

Two Shortcuts

Here are two shortcuts which will save time in typing programs.

1. It's not necessary to include the word **LET** in a **LET** statement! The statement

   ```
   10 A=5
   ```

   means the same thing to the computer as

   ```
   10 LET A=5
   ```

2. A question mark may be used in place of the word **PRINT.** Therefore, the statement

   ```
   10 ? A, A$
   ```

means the same thing as the statement

```
10 PRINT A,A$
```

TEST YOUR UNDERSTANDING 1 (answer on page 51)

What is the output of the following program?

```
10 A=3: B=7
20 A=2*B+3*A
30 ? A,B^2
40 END
```

Using a Printer

In writing programs and analyzing their output, it's often easier to rely on written output rather than output on the screen. In computer terminology, written output is called **hard copy** and may be provided by a wide variety of printers. Your Franklin computer may be attached to a large number of such printers, ranging from a dot-matrix thermal printer costing only a few hundred dollars to a daisy wheel printer costing several thousand dollars. As you begin to make serious use of your computer, you'll find it difficult to do without hard copy.

Indeed, writing programs is much easier if you can consult a hard copy listing of your program at various stages of program development. (One reason is that in printed output you are not confined to looking at your program in 10-14line "snapshots.") Also, you'll want to use the printer to produce output of programs, ranging from tables of numerical data to address lists and text files produced via a word processing program.

To use a printer, it's necessary to install a printer interface card in one of the unused slots (#1 - #7). Slot #1 is usually used for the printer. To use the printer, you must first direct output to the printer via the statement

```
10 PR#1
```

As given above, the statement is for use within a program. All subsequent output to the screen will also be printed on the printer. If you are in command mode (no program running), you may type **PR#1** followed by RETURN (without a line number). All subsequent output will then go to both the screen and the printer.

Now, you may produce hard copy on your printer by using the Floating Point BASIC statement **PRINT.** For example, the statement

```
10 PRINT A,A$
```

will print the current values of A and A$ on the printer, in print fields 1 and 2. (As is the case with the screen, Floating Point BASIC divides the printer line into print fields which are 16 columns wide.) Moreover, the statement

```
20 PRINT "Customer","Credit Limit","Most Recent Pchs"
```

will result in printing three headings in the first three print fields, namely:

```
Customer          Credit Limit          Most Recent Pchs
```

To return output to the screen only, use the command

```
PR#0
```

Printing on the printer proceeds very much like printing on the screen. It's important to realize, however, that after the **PR#1** command is given, output will be printed on both the screen and the printer.

Some Things to Check

Writing programs in Floating Point BASIC isn't difficult. However, it does require a certain amount of care and meticulous attention to detail. Each person must develop an individual programming style.

Here are a few tips which may help the novice programmer over some of the rough spots of writing those first few programs.

1. Carefully think your program through. Break up the computation into steps. Outline the programming necessary for each of the steps.
2. Work through your program by hand, pretending that you are the computer. Don't rush. Go through your program one step at a time and check to make sure that it does what you want it to do.
3. Have you given all variables the values you want? Remember, if you don't specify the value of a variable, Floating Point BASIC will automatically assign it the value 0. This may not be the value you intend!
4. Are all your loops complete? That is, have you included a **NEXT** corresponding to each **FOR?** This is an easy mistake to make, but it is also easy to catch. If Floating Point BASIC doesn't find a **NEXT** corresponding to a **FOR** when it attempts to run the program, it will report the mistake and the line number in which it occurs. This is just one of a series of checks which Floating Point BASIC makes for consistency and completeness. (Later, you'll see some of the various error messages which Floating Point BASIC can provide.) **FOR ... NEXT** loops may be contained in one another. (They may be nested.) But the loop which starts earlier must end later. In other words, loops may not "cross" one another.
5. Check to see that your **IF ... THEN** statements don't create any infinite loops. This may be a difficult error to spot. However, it can be located with the following check. When you go back to an earlier part of the program, ask yourself: What condition must be present if the program

is not to keep doubling back forever? Is this condition guaranteed to occur?

In the upcoming chapters you'll find further ideas on debugging your programs and on programming technique. For now, however, let's move on with learning to make the computer do interesting things!

ANSWERS TO TEST YOUR UNDERSTANDING 1

```
1:  23          49
```

3
More About Floating Point BASIC

3.1 Working With Tabular Data

In the preceding chapter, you found the notion of a variable and used variable names like

AA, B1, CZ, W0

Unfortunately, the supply of variables available to us is not sufficient for many programs. Indeed, there are relatively innocent programs which require hundreds or even thousands of variables. To meet the needs of such programs, Floating Point BASIC allows the use of so-called **subscripted variables.** Such variables are used constantly by mathematicians and are identified by numbered subscripts attached to a letter. For instance, here is a list of 1000 variables as they might appear in a mathematical work:

$A_1, A_2, A_3, \ldots, A_{1000}$

The numbers used to distinguish the variables are called subscripts. Likewise, the Floating Point BASIC language allows definition of variables to be distinguished by subscripts. However, since the computer has difficulty placing the numbers in the traditional position, they're placed in parentheses on the same line as the letter. For example, the above list of 1000 different variables would be written in Floating Point BASIC as

A(1),A(2),A(3),...,A(1000)

Please note that the variable A(1) isn't the same as the variable A1. You may use both of them in the same program and Floating Point BASIC will interpret them as different.

A subscripted variable is really a group of variables with a common letter identification which are distinguished by different integer "subscripts." For instance, the above group of variables would constitute the subscripted variable A( ). It's often useful to view a subscripted variable as a table or array. For example, the subscripted variable A( ) considered above can be viewed as providing the following table of information:

A(1)
A(2)
A(3)
.
.
.
A(1000)

As shown here, the subscripted variable defines a table consisting of 1000 rows. Row number J contains a single entry, namely, the value of the variable A(J). The first row contains the value of A(1), the second the value of A(2), and so forth. Since a subscripted variable can be thought of as a table (or array), subscripted variables are often called **arrays.**

The array shown is a table consisting of 1000 rows and a single column. The Franklin allows you to consider more general arrays. For example, consider the following financial table which records the monthly income for January, February, and March from each of a chain of four dry cleaning stores:

	Store #1	Store #2	Store #3	Store #4
Jan.	1258.38	2437.46	4831.50	987.12
Feb.	1107.83	2045.68	3671.86	1129.47
March	1298.00	2136.88	4016.73	1206.34

This table has three rows and four columns. Its entries may be stored in the computer as a set of 12 variables:

```
A(1,1) A(1,2) A(1,3) A(1,4)
A(2,1) A(2,2) A(2,3) A(2,4)
A(3,1) A(3,2) A(3,3) A(3,4)
```

This array of variables is very similar to a subscripted variable, except that there are now two subscripts. The first subscript indicates the row number and the second subscript indicates the column number. For example, the variable A(3,2) is in the third row, second column. A collection of variables such as that given above is called a **two-dimensional array** or a **doubly-subscripted variable.** Each setting of the variables in such an array defines a tabular array. For example, if you assigned the values

```
A(1,1)=1258.38, A(1,2)=2437.46,
A(1,3)=4831.90,
```

and so forth, then you'd have the table of earnings from the dry cleaning chain.

So far, you've only considered numeric arrays—arrays whose variables can assume only numerical values. However, it's possible to have arrays with variables that assume string values. (Recall that a string is a sequence of characters: letter, numeral, punctuation mark, or other printable keyboard symbol.) For example, here's an array which can contain string data:

```
A$(1)
A$(2)
A$(3)
A$(4)
```

Here the dollar signs indicate that each of the variables of the array is a string variable. If you assign the values

```
A$(1)="SLOW", A$(2)="FAST", A$(3)="FAST", A$(4)="STOP"
```

then the array is just the table of words

```
SLOW
FAST
FAST
STOP
```

Similarly, the employee record table

```
Social Security Number   Age   Sex

      178654775          38    M
      345861023          29    F
      789257938          34    F
      3/5486595          42    M
      457696064          21    F
```

may be stored in an array of the form B$(I,J), where I assumes any one of the values 1,2,3,4,5 (I is the row), and J assumes any one of the values 1,2,3 (J = the column). For example, B$(1,1) has the value "178654775", B$(1,2) has the value "38", B$(1,3) has the value "M", and so forth.

The Franklin computer even allows you to have arrays which have three, four, or even more subscripts. For example, consider the dry cleaning chain array introduced above. Suppose that you had one such array for each of ten consecutive years. This collection of data could be stored in a three-dimensional array of the form C(I,J,K), where I and J represent the row and column, just as before, and K represents the year. (K could assume the values 1,2,3,...,10.)

An array may involve any number of dimensions up to 88. The subscripts corresponding to each dimension may assume values from 0 to 32767. For all practical applications, any size array is permissible.

You must inform the computer of the sizes of the arrays you plan to use in a program. This allows the computer to allocate memory space to house all the values. To specify the size of an array, use a dimension **(DIM)** statement. For

example, to define the size of the subscripted variable A(J) , J = 1,...,1000, you insert the statement

```
10 DIM A(1000)
```

in the program. This statement informs the computer to expect variables A(0),A(1),...,A(1000) in the program, and that it should set aside memory space for 1001 variables. Note that, in the absence of further instructions from you, Floating Point BASIC begins all subscripts at 0. If you wish to use A(0), fine. If not, ignore it.

You need not use all the variables defined by a **DIM** statement. For example, in the case of the **DIM** statement above, you might actually use only the variables A(1),...,A(900). Don't worry about it! Just make sure that you've defined enough variables. Otherwise you could be in trouble. For example, in the case of the subscripted variable above, your program might make use of the variable A(1001). This will create an error condition. Suppose that this variable is used first in line 570. When you attempt to run the program, the computer will report:

```
? BAD SUBSCRIPT ERROR
```

Moreover, execution of the program will be halted. To fix the error, merely redo the **DIM** statement to accommodate the undefined subscript.

To define the size of a two-dimensional array, use a **DIM** statement of the form

```
10 DIM A(5,4)
```

This statement defines an array A(I,J), where I can assume the values 0,1,2,3,4,5 and J can assume the values 0,1,2,3,4. Arrays with three or more subscripts are defined similarly.

TEST YOUR UNDERSTANDING 1 (answers on page 59)

Here's an array.

```
12  645.80
148 489.75
589 12.89
487 14.50
```

a. Define an appropriate subscripted variable to store this data.
b. Define an appropriate **DIM** statement.

It's possible to dimension several arrays with one **DIM** statement. For example, the dimension statement

```
10 DIM A(1000), B$(5), A(5,4)
```

defines the array A(0),...,A(1000), the string array B$(0),...,B$(5), and the two-dimensional array A(I,J) , I = 0,...,5; J = 0,...,4.

You know how to set aside memory space for the variables of an array. Now it's time to take up the problem of assigning values to these variables. You could use individual **LET** statements, but with 1000 variables in an array, this could lead to an unmanageable number of statements. There are more convenient methods which make use of loops. The next two examples illustrate two of these methods.

Example 1. Define an array A(J) , J = 1,2,...,1000 and assign the following values to the variables of the array:

```
A(1)=2, A(2)=4, A(3)=6, A(4)=8, ...
```

Solution. You wish to assign each variable a value equal to twice its subscript. That is, you wish to assign A(J) the value 2*J. To do this, use a loop:

```
10 DIM A(1000)
20 FOR J=1 TO 1000
30 A(J)=2*J
40 NEXT J
50 END
```

Note that the program ignores the variable A(0). Like any variable which has not been assigned a value, it has the value 0.

TEST YOUR UNDERSTANDING 2 (answer on page 59)

Write a program which assigns the variables A(0),...,A(30) the values A(0) = 0, A(1) = 1, A(2) = 4, A(3) = 9, ...

When the computer is first turned on or is reset, all variables (including those in arrays) are cleared. All numeric variables are set equal to 0, and all string variables are set equal to the null string (the string with no characters in it). If you wish to return all variables to this state during the execution of a program, use the command **CLEAR.** For example, when the computer encounters the command

```
570 CLEAR
```

it will reset all the variables. The **CLEAR** command can be convenient if, for example, you wish to use the same array to store two different sets of information at two different stages of the program. After the first use of the array you could then prepare for the second use by executing a **CLEAR.**

Example 2. Define an array corresponding to the employee record table above. Input the values given and print the table on the screen.

Solution. Your program will print the headings of the columns and then ask for the table entries, one row at a time. Store the entries in the array B$(I,J), where I is one of 1,2,3,4,5 and J is one of 1,2,3,4. Dimension the array as B$(5,4).

```
5 DIM B$(5,4)
10 PRINT "SOC.SEC. #", "AGE", "SEX"
20 FOR I=1 TO 5
30 INPUT "SS #,AGE,SEX"; B$(I,1),B$(I,2),B$(I,3)
40 PRINT B$(I,1),B$(I,2),B$(I,3)
50 NEXT I
60 END
```

TEST YOUR UNDERSTANDING 3 (answer on page 59)

Suppose that your program uses a 9 x 2 array A$(I,J), a 9 x 1 array B$(I,J), and a 9 x 5 array C(I,J). Suppose that the strings involved are at most 10 characters in length. Write an appropriate DIM statement(s).

If you plan to dimension an array, you should always insert the **DIM** statement before the variable first appears in your program. Otherwise, the first time Floating Point BASIC comes across the array, it will assume that the subscripts go from 0 to 10. If it subsequently comes across a **DIM** statement, it will think you're changing the size of the array in the midst of the program, something which isn't allowed. If you try to change the size of an array in the middle of a program, you'll get an error:

```
? REDIM'D ARRAY ERROR
```

Exercises (answers on page 201)

For each of the following tables, define an appropriate array and determine the appropriate DIM statement.

1. 5
   2
   1.7
   4.9
   11
2. 1.1 2.0 3.5
   1.7 2.4 6.2
3. JOHN
   MARY
   SIDNEY
4. 1 2 3

5.

RENT	575.
UTILITIES	249.78
CLOTHES	174.98
CAR	348.7

6. Display the following array on the screen:

	Receipts Store #1	Store #2
1/1-1/10	57385.48	89485.45
1/11-1/20	39485.98	76485.49
1/21-1/31	45467.21	71494.25

7. Write a program that displays the array of Exercise 6 along with totals of the receipts from each store.
8. Expand the program in Exercise 7 so that it calculates and displays the totals of ten-day periods. (Your screen will not be wide enough to display the ten-day totals, so display them in a separate array.)
9. Devise a program which keeps track of the inventory of an appliance store chain. Store the current inventory in an array of the form

	Store #1	Store #2	Store #3	Store #4
Refrig.				
Stove				
Vacuum				
Air Cond.				
Disposal				

Your program should:(1) input the inventory corresponding to the beginning of the day, (2) continually ask for the next transaction—the store number and the number of appliances of each item sold, and (3) in response to each transaction, update the inventory array and redisplay it on the screen.

ANSWERS TO TEST YOUR UNDERSTANDING 1, 2, and 3

1: a.
```
A(I,J), I=1,2,3,4; J=1,2
```
b.
```
DIM A(4,2)
```
2:
```
10 DIM A(30)
20 FOR J=0 TO 30
30 A(J)=J^2
40 NEXT J
50 END
```
3:
```
DIM A$(9,2),B$(9,1),C(9,5)
```

3.2 Inputting Data

The preceding section introduced arrays and discussed several methods for assigning values to the variables of an array. The most flexible method was via the **INPUT** statement. However, this can be a tedious method for large arrays. Fortunately, Floating Point BASIC provides an alternate method for inputting data.

A given program may need many different numbers and strings. You may store the data needed in one or more **DATA** statements. A typical data statement has the form

```
10 DATA 3.457, 2.588, 11234, "WINGSPAN"
```

Note that this data statement consists of four data items, three numeric and one string. The data items are separated by commas. You may include as many data items in a single **DATA** statement as the line allows. Moreover, you may include any number of **DATA** statements in a program and they may be placed anywhere in the program, although a common placement is at the end of the program (just before the **END** statement). Note that you enclosed the string constant "WINGSPAN" in quotation marks. Actually, this isn't necessary. A string constant in a **DATA** statement doesn't need quotes as long as the string doesn't start with a blank.

The **DATA** statements may be used to assign values to variables and, in particular, to variables in arrays. Here's how. In conjunction with the **DATA** statements, you use one or more **READ** statements. For example, suppose that the above **DATA** statement appeared in a program. Further, suppose that you wish to assign the values

```
A=3.457, B=2.588, C=11234, Z$="WINGSPAN"
```

This can be accomplished via the **READ** statement

```
100 READ A,B,C,Z$
```

Here's how the **READ** statement works. On encountering a **READ** statement, the computer will look for a **DATA** statement. It will then assign values to the variables in the **READ** statement by taking the values, in order, from the **DATA** statement. If there is insufficient data in the first **DATA** statement, the computer will continue to assign values using the data in the next **DATA** statement. If necessary, the computer will proceed to the third **DATA** statement, and so forth.

TEST YOUR UNDERSTANDING 1 (answer on page 66)

Assign the following values:
A(1) = 5.1, A(2) = 4.7, A(3) = 5.8, A(4) = 3.2, A(5) = 7.9, A(6) = 6.9.

The computer maintains an internal pointer which points to the next **DATA** item to be used. If the computer encounters a second **READ** statement, it will start reading where it left off. For example, suppose that instead of the above **READ** statement, you use the two read statements

```
100 READ A,B
200 READ C,Z$
```

Upon encountering the first statement, the computer will look for the location of the pointer. Initially, it will point to the first item in the first **DATA** statement. The computer will assign the values A = 3.457 and B = 2.588. Moreover, the position of the pointer will be advanced to the third item in the **DATA** statement. Upon encountering the next **READ** statement, the computer will assign values beginning with the one designated by the pointer, namely C=11234 and Z$="WINGSPAN".

TEST YOUR UNDERSTANDING 2 (answer on page 66)

What values are assigned to A and B$ by the following program?

```
10 DATA 10,30,"ENGINE", "TACH"
20 READ A,B
30 READ C$,B$
40 END
```

The following example illustrates the use of **DATA** statements in assigning values to an array.

Example 1. Suppose that the monthly electricity costs of a certain family are as follows:

Jan.	$89.74	Feb.	$95.84	March	$79.42
Apr.	78.93	May	72.11	June	115.94
July	158.92	Aug.	164.38	Sep.	105.98
Oct.	90.44	Nov.	89.15	Dec.	93.97

Write a program calculating the average monthly cost of electricity.

Solution. Let's unceremoniously dump all of the numbers shown above into **DATA** statements at the end of the program. Arbitrarily, let's start the **DATA** statements at line 1000, with **END** at 2000. This affords us plenty of room. To calculate the average, you must add up the numbers and divide by 12. To do this, you first create an array A(J), J = 1,2,...,12 and set A(J) equal to the cost of electricity in the Jth month. Do this via a loop and the **READ** statement. Then use a loop to add all the A(J)s. Finally, divide by 12 and **PRINT** the answer. Here's the program.

```
10 DIM A(12)
15 REM LINES 20-40 ASSIGN VALUES TO A(J)
```

```
20 FOR J=1 TO 12
30 READ A(J)
40 NEXT J
50 FOR J=1 TO 12
60 C=C+A(J): REM   C ACCUMULATES THE SUM OF THE A(J)
70 NEXT J
80 C=C/12 : REM DIVIDE SUM BY 12
90 PRINT "THE AVERAGE MONTHLY COST OF ELECTRICITY IS",C
1000 DATA 89.74, 95.84, 79.42, 78.93, 72.11, 115.94
1010 DATA 158.92, 164.38, 105.98, 90.44, 89.15, 93.97
2000 END
```

The following program could be helpful in preparing the payroll of a small business.

Example 2. A small business has five employees. Here are their names and hourly wages.

Name	Hourly Wage
Joe Polanski	7.75
Susan Greer	8.50
Allan Cole	8.50
Betsy Palm	6.00
Herman Axler	6.00

Write a program which accepts as input hours worked for the current week and calculates the current gross pay and the amount of Social Security tax to be withheld from their pay. (Assume that the Social Security tax amounts to 6.70 percent of gross pay.)

Solution. Let's keep the hourly wage rates and names in two arrays, called A(J) and B$(J), respectively, where J = 1,2,3,4,5. Note that you can't use a single two-dimensional array for this data since the names are string data, and the hourly wage rates are numerical. (Recall that Floating Point BASIC does not allow us to mix the two kinds of data in an array.) The first part of the program will be to assign the values to the variables in the two arrays. Next, the program will, one by one, print out the names of the employees and ask for the number of hours worked during the current week. This data will be stored in the array C(J), J = 1,2,3,4,5. The program will then compute the gross wages as A(J)*C(J) (that is, <wage rate> times <number of hours worked>). This piece of data will be stored in the array D(J), J = 1,2,3,4,5. Next, the program will compute the amount of Social Security tax to be withheld as .0670*D(J). This piece of data will be stored in the array E(J), J = 1,2,3,4,5. Finally, all the computed data will be printed on the screen. Here's the program:

```
10 DIM A(5),B$(5),C(5),D(5),E(5)
20 FOR J=1 TO 5
30 READ B$(J),A(J)
```

```
40 NEXT J
50 FOR J=1 TO 5
60 PRINT "TYPE CURRENT HOURS OF ", B$(J)
70 INPUT C(J)
80 D(J)=A(J)*C(J)
90 E(J)=.067*D(J)
100 NEXT J
110 PRINT "EMPLOYEE","GROSS WAGES","SS.TX"
120 FOR J=1 TO 5
130 PRINT B$(J),D(J),E(J)
140 NEXT J
200 DATA JOE POLANSKI, 7.75, SUSAN GREER, 8.5
210 DATA ALLAN COLE, 8.5, BETSY PALM, 6.
220 DATA HERMAN AXLER, 6.
1000 END
```

In certain applications, you may wish to read the same **DATA** statements more than once. To do this you must reset the pointer via the **RESTORE** statement. For example, consider the following program.

```
10 DATA 2.3, 5.7, 4.5, 7.3
20 READ A,B
30 RESTORE
40 READ C,D
50 END
```

Line 20 sets A equal to 2.3 and B equal to 5.7. The **RESTORE** statement of line 30 moves the pointer back to the first item of data, 2.3. The **READ** statement of line 40 then sets C equal to 2.3 and D equal to 5.7. Note that without the **RESTORE** in line 30, the **READ** statement in line 40 would set C equal to 4.5 and D equal to 7.3.

There are two common errors in using **READ** and **DATA** statements. First, you may instruct the program to **READ** more data than is present in the **DATA** statements. For example, consider the following program.

```
10 DATA 1,2,3,4
20 FOR J=1 TO 5
30 READ A(J)
40 NEXT J
50 END
```

This program attempts to read five pieces of data, but the **DATA** statement only has four. In this case, you'll receive an error message

```
? OUT OF DATA ERROR IN 30
```

A second common error is attempting to assign a string value to a numeric variable or vice versa. Such an attempt will lead to

```
? TYPE MISMATCH ERROR
```

Exercises (answers on page 202)

Each of the following programs assigns values to the variables of an array. Determine which values are assigned.

1.
```
10 DIM A(10)
20 FOR J=1 TO 10
30 READ A(J)
40 NEXT J
50 DATA 2,4,6,8,10,12,14,16,18,20
100 END
```

2.
```
10 DIM A(3),B(3)
20 FOR J=0 TO 3
30 READ A(J), B(J)
40 NEXT J
50 DATA 1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9
60 END
```

3.
```
10 DIM A(3),B$(3)
20 FOR J=0 TO 3
30 READ A(J)
40 NEXT J
50 FOR J=0 TO 3
60 READ B$(J)
70 NEXT J
80 DATA 1,2,3,4,A,B,C,D
90 END
```

4.
```
10 DIM A(3), B(3)
20 READ A(0),B(0)
30 READ A(1),B(1)
40 RESTORE
50 READ A(2),B(2)
60 READ A(3),B(3)
70 DATA 1,2,3,4,5,6,7,8
80 END
```

5.
```
10 DIM A(3,4)
20 FOR I=1 TO 3
30 FOR J=1 TO 4
40 READ A(I,J)
50 NEXT J
60 NEXT I
70 DATA 1,2,3,4,5,6,7,8,9,10,11,12
80 END
```

6.
```
10 DIM A(3,4)
20 FOR J=1 TO 4
30 FOR I=1 TO 3
40 READ A(I,J)
50 NEXT I
```

```
60 NEXT J
70 DATA 1,2,3,4,5,6,7,8,9,10,11,12
80 END
```

Each of the following programs contains an error. Find it.

7.
```
10 DIM A(5)
20 FOR J=1 TO 5
30 READ A(J)
40 NEXT J
50 DATA 1,2,3,4
60 END
```

8.
```
10 DIM A(5)
20 FOR J=1 TO 5
30 READ A(J)
40 NEXT J
50 DATA 1,A,2,B
60 END
```

9. Here's a table of Federal Income Tax Withholding of weekly wages for an individual claiming one exemption. Assume that each of the employees, in the business discussed in the text, claims a single exemption. Modify the program given so that it correctly computes Federal Withholding and the net amount of wages. (That is, the total after Federal Withholding and Social Security are deducted.)

Wages at Least	But Less Than	Tax Withheld
200	210	29.10
210	220	31.20
220	230	31.80
230	240	36.40
240	250	39.00
250	260	41.60
260	270	44.20
270	280	46.80
280	290	49.40
290	300	52.10
300	310	55.10
310	320	58.10
320	330	61.10
330	340	64.10
340	350	67.10

10. Here's a set of 24 hourly temperature reports as compiled by the National Weather Service. Write a program to compute the average temperature for the last 24 hours. Let your program respond to a query concerning the temperature at a particular hour. (For example, what was the temperature at 2:00 PM?)

	AM	PM
12:00	10	38
1:00	10	39
2:00	9	40

3:00	9	40
4:00	8	42
5:00	11	38
6:00	15	33
7:00	18	27
8:00	20	22
9:00	25	18
10:00	31	15
11:00	35	12

ANSWERS TO TEST YOUR UNDERSTANDING 1 and 2

```
1:  10 DATA 5.1,4.7,5.8,3.2,7.9,6.9
    20 FOR J=1 TO 6
    30 READ A(J)
    40 NEXT J
    50 END
2:  A=10, B$="TACH"
```

3.3 Advanced Printing

In this section, you see how you can format output on the screen and on the printer. Floating Point BASIC is quite flexible in the form in which you can cast output. You have control over placement of output on the line, the degree of accuracy to which calculations are displayed, and so forth. Let's begin by reviewing what you've already learned about printing.

There are 40 print positions in each line. These are divided into two print zones of 16 characters each and one zone with 8 characters. To start printing at the beginning of the next print zone, insert a comma between the items to be printed. To avoid any space between items, separate them in the **PRINT** statement by a semicolon. For example, the following program

```
10 A=5
20 PRINT "THE VALUE OF A IS EQUAL TO ";A
30 END
```

will result in the following screen display:

```
THE VALUE OF A IS EQUAL TO 5
```

Note that the first print item ends with a space. This is to guarantee a space between the final O and the 5.

TEST YOUR UNDERSTANDING 1 (answer on page 71)

Write a program which allows you to input two numbers. The program should then display them as an addition problem in the form 5 + 7 = 12.

Horizontal Tabbing

You may begin a print item in any position on a line. To do this, use the **TAB** command. Note that a logical line may be up to 255 characters long. An oversized line will wrap around to the next line. The positions of characters on logical lines are numbered from 0 to 255. The statement **TAB(7)** means to move to position 7. **TAB** is always used in conjunction with a **PRINT** statement. For example, the **PRINT** statement

```
50 PRINT TAB(7) A
```

will print the value of the variable A, beginning in print position 7. It's possible to use more than one **TAB** per **PRINT** statement. For example, the statement

```
100 PRINT TAB(5) A; TAB(15) B
```

will print the value of A beginning in print position 5, and the value of B beginning in print position 15. Note the semicolon between the two **TAB** instructions.

In typing a **PRINT** statement, you may run out of room on the line. To get around this problem, end the **PRINT** statement with a semicolon and continue the list of print items in another **PRINT** statement on the next line. For example, consider the pair of statements

```
100 PRINT "INVENTORY";
110 PRINT , "OF LADIES SHOES"
```

The first line has a single print item. The semicolon indicates continued printing on the same line. The comma which begins the second **PRINT** statement moves printing to the beginning of the next print zone, where the string in line 110 is printed. Here's what the output looks like:

```
INVENTORY                    OF LADIES SHOES
```

TEST YOUR UNDERSTANDING 2 (answer on page 71)

Write an instruction printing the value of A in column 25 and the value of B seven columns further to the right.

Formatting Numbers

So far, numerical output has been accepted in the form in which it was dispensed from the computer. However, this has led to occasional embarrassments earlier in this book. For example, columns of numbers didn't line up, dollars and cents items were displayed to hundredths of a cent, and so forth. Let's now consider how to control the format of numerical output.

The Franklin carries out calculations to as many as 10 significant digits. Here are some typical examples of output:

```
1001, 1075.312, 123456789, 1.24159211E12
```

In many applications, it's necessary to round a number X to a given number of decimal places, say N places. This may be accomplished in Floating Point BASIC using the statement

```
INT(X*10^N+.5)/10^N
```

(For now, don't worry about the derivation of this statement.) For example, 5.75% rounded to one decimal place is equal to

```
INT(5.75#*10+.5)/10
```

In preparing financial statements, it's usually necessary to print columns of numbers which are aligned, the one's under the one's column, the ten's under the ten's column, and so forth. This may be done by printing an appropriate number of blanks in front of the number. For example, suppose you wish to print a column of numbers, the longest of which consists of 9 digits and a decimal point. At the beginning of each number in the column, you would insert enough spaces so that the length of the resulting number would be 10 characters (including the decimal point). The "length" of a number X may be computed using the statement:

```
LEN(STR$(X))
```

(The **STR$(X)** is just X converted to a string constant. The **LEN** gives the length of the string.) The number of spaces to be inserted would then be

```
10 - LEN(STR$(X))
```

Finally, this number of spaces in front of X may be generated by using the **PRINT** statement:

```
PRINT TAB(10 - LEN(STR$(X)));X
```

TEST YOUR UNDERSTANDING 3 (answer on page 71)

Write an instruction which prints the number 456.75387 rounded to two decimal places.

TEST YOUR UNDERSTANDING 4 (answer on page 71)

Write a program to calculate and display the numbers 2^J, J = 1,2,3,...,15. The columns of the numbers should be properly aligned on the right.

Example 1. Here's a list of checks written by a family during the month of March.

$15.32, $387.25, $57.98, $3.47, $15.88

Print the list of checks on the screen with the columns properly aligned and the total displayed below the list of check amounts, in the form of an addition problem.

Solution. First read the check amounts into an array A(J), J = 1,2,3,4,5. While you read the amounts, you accumulate the total in the variable B. Use a second loop to print the display in the desired format.

```
10 DATA 15.32, 387.25, 57.98, 3.47, 15.88
20 FOR J=1 TO 5
30 READ A(J)
40 B=B+A(J)
50 PRINT "$";SPC(6-LEN(STR$(A(J)))); A(J)
60 NEXT J
70 PRINT "--------"
80 PRINT "$";SPC(6-LEN(STR$(A(J)))); B
90 END
```

Here's what the output will look like:

```
$ 15.32
$387.25
$ 57.98
$  3.47
$ 15.88

$479.90
```

Note that line 70 is used to print the line under the column of figures.

Exercises (answers on page 204)

Write programs which generate the following displays. The lines of dots are not included for you to display. They're furnished for you to judge spacing.

1. THE AREA OF A IS 5.375

   .....

2.
```
THE VALUE OF X IS      5.378
.............................
```

3.
```
DATE  QTY   @    COST     DISCOUNT     COST
.........................................
```

4.
```
   6.753
  15.111
 111.85
   6.702
-------
          Calculate
          Sum
```

5.
```
$ 12.82
$117.58
$  5.87
$   .99
-------
          Calculate
          Sum
```

6.
```
                                              Date 3/18/81

Pay to the Order of Wildcatters, Inc.

The sum of      **********$89,385.00
```

7.
```
    5,787
      387
127,486
 38,531
-------
          Calculate
          Sum
```

8.
```
   $385.41
 - $17.85
  -------
            Calculate
            Difference
```

9. Write a program which rounds a number to the nearest integer. For example, if the input is 11.7, the output is 12. If the input is 158.2, the output is 158. Your program should accept the number to be rounded via an **INPUT** statement.
10. Write a program which allows your computer to function as a cash register. Let the program accept purchase amounts via **INPUT** statements. Let the user tell the program when the list of **INPUT**s is complete. The program should then print out the purchase amounts, with

dollar signs and columns aligned, compute the total purchase, add 5 percent sales tax, compute the total amount due, ask for the amount paid, and compute the change due.

ANSWERS TO TEST YOUR UNDERSTANDING 1,2,3, and 4

```
1:  10 INPUT A,B
    20 PRINT A;"+";B;"=";A+B
    30 END
2:  10 PRINT TAB(25) A;TAB(32) B
3:  10 PRINT INT(456.75387*10^2+.5)/10^2
4:  10 FOR J=1 TO 15
    20 PRINT SPC(5-LEN(STR$(2^J)));2^J
    30 NEXT J
    40 END
```

3.4 Gambling With Your Computer

One of the most interesting features of your computer is its ability to generate events whose outcomes are "random." For example, you may instruct the computer to "throw a pair of dice" and produce a random pair of integers between 1 and 6. You may instruct the computer to "pick a card at random from a deck of 52 cards." You may also program the computer to choose a two-digit number "at random." And so forth. The source of all such random choices is the **random number generator,** which is a part of Floating Point BASIC. Let's begin by explaining what the random number generator is and how to access it. You'll then find a number of interesting applications involving computer-assisted instruction and games of chance.

You may generate random numbers using the Floating Point BASIC function **RND(1).** To explain how this function works, let's consider the following program:

```
10 FOR X=1 TO 500
20 PRINT RND(1)
30 NEXT X
40 END
```

This program consists of a loop which prints 500 numbers, each called **RND(1).** Each of these numbers lies between 0.0000000000 (inclusive) and 1.000000000 (exclusive). Each time **RND(1)** is called (as in line 20 above), the computer makes a "random" choice from among the numbers in the indicated range. This is the number that is printed.

To get a better idea of what this is all about, you should generate some random numbers using a program like the one above. Unless you have a printer, 500 numbers will be too many for you to look at in one viewing. You should print two random numbers on one line (one per print zone) and limit yourself to 24 displayed lines at one time. Here's a partial printout of such a program.

```
.2451213213     .3050034958
.9845468610     .9011598475
.8966099342     .6602124859
.5839383519     .4481638341
.1371193105     .2265442384
```

What makes these numbers "random" is that the procedure the computer uses to select them is "unbiased," with all numbers having an equal likelihood of selection. Moreover, if you generate a large collection of random numbers, then numbers between 0 and .1 will comprise approximately 10 percent of those chosen, those between .5 and 1.0 will comprise 50 percent of those chosen, and so forth. In some sense, the random number generator provides a uniform sample of the numbers between 0 and 1.

TEST YOUR UNDERSTANDING 1 (answer on page 76)

Assume that RND(1) is used to generate 1000 numbers. Approximately how many of these numbers would you expect to lie between .6 and .9?

The function **RND(1)** generates random numbers lying between 0 and 1. In many applications, however, you'll require randomly chosen **integers** lying in a certain range. For example, suppose that you wish to generate random integers chosen from among 1,2,3,4,5,6. Let's multiply **RND(1)** by 6, to obtain **6*RND(1).** This is a random number between 0.0000000000 and 5.999999999. Next, let's add 1 to this number. Then 6*RND(1) + 1 is a random number between 1.000000000 and 6.999999999. To obtain integers from among 1,2,3,4,5,6, you must "chop off" the decimal portion of the number 6*RND(1) + 1. To do this, use the **INT** function. If X is any number, then INT(X) is the largest integer less than or equal to X. For example,

```
INT(5.23)=5, INT(7.99)=7, INT(100.001)=100
```

Be careful in using INT with negative X. The definition given is correct, but unless you think things through, it's easy to make an error. For example,

```
INT(-7.4)=-8
```

since the largest integer less than or equal to -7.4 is equal to -8. (Draw -7.4 and -8 on a number line to see the point!) Let's get back to our random numbers. To chop off the decimal portion of 6*RND(1) + 1, compute

INT(6*RND(1)+1). This last expression is a random number from among 1,2,3,4,5,6. Similarly, the expression

```
INT(100*RND(1)+1)
```

may be used to generate random numbers from among the integers 1,2,3,...,100.

TEST YOUR UNDERSTANDING 2 (answer on page 76)

Generate random integers from 0 to 1. (This is the computer analogue of flipping a coin: 0 = heads, 1 = tails.) Run this program to generate 50 coin tosses. How many heads and how many tails occur?

Example 1. Write a program which turns the computer into a pair of dice. Your program should report the number rolled on each as well as the total.

Solution. Hold the value of die #1 in the variable X and the value of die #2 in variable Y. The program will compute values for X and Y and print out the values and the total X + Y.

```
10 HOME
20 LET X=INT(6*RND(1)+1)
30 LET Y=INT(6*RND(1)+1)
40 PRINT "LADIES AND GENTLEMEN, BETS PLEASE!"
50 INPUT "ARE ALL BETS DOWN(Y/N) "; A$
60 IF A$="Y" THEN 100
70 GOTO 40
100 PRINT "THE ROLL IS ",X,Y
110 PRINT "THE WINNING TOTAL IS " ; X+Y
120 INPUT "PLAY AGAIN(Y/N) "; B$
130 IF B$="Y" THEN 10
200 PRINT "THE CASINO IS CLOSING. SORRY!"
210 END
```

Note the use of computer-generated conversation on the screen. Note also how the program uses lines 120–130 to allow the player to control how many times the game will be played.

TEST YOUR UNDERSTANDING 3 (answer on page 76)

Write a program which flips a "biased coin". Let it report "heads" one-third of the time and "tails" two-thirds of the time.

You may enhance the realism of a gambling program by letting the computer keep track of bets as in the following example.

Example 2. Write a program which turns the computer into a roulette wheel. Let the computer keep track of bets and winnings for up to five players. For simplicity, assume that the only bets are on single numbers. (In the next section, you'll be able to remove this restriction!)

Solution. A roulette wheel has 38 positions: 1-36, 0, and 00. Represent these as the numbers 1-38, with 37 corresponding to 0 and 38 corresponding to 00. A spin of the wheel will consist of choosing a random integer between 1 and 38. The program will start by asking for the number of players. For a typical spin of the wheel, the program will ask for bets by each player. A bet will consist of a number (1-38) and an amount bet. The wheel will then spin. The program will determine the winners and losers. A payoff for a win is 32 times the amount bet. Each player has an account stored in an array A(J), J = 1,2,3,4. At the end of each spin, the accounts are adjusted and displayed. Just as in Example 1 above, the program asks if another play is desired. Here's the program.

```
10 INPUT "NUMBER OF PLAYERS ";N
20 DIM A(5),B(5),C(5): REM AT MOST 5 PLAYERS ALLOWED
25 REM LINES 30-60 ALLOW PLAYERS TO PURCHASE CHIPS
30 FOR J=1 TO N : REM FOR EACH OF THE PLAYERS
40 PRINT "PLAYER "; J
50 INPUT "HOW MANY CHIPS "; A(J)
60 NEXT J
100 PRINT "LADIES AND GENTLEMEN!"
105 PRINT "PLACE YOUR BETS PLEASE!"
110 FOR J=1 TO N : REM FOR EACH OF THE PLAYERS
120 PRINT "PLAYER "; J
130 INPUT "NUMBER, AMOUNT "; B(J),C(J):REM INPUT BET
140 NEXT J
200 X=INT(38*RND(1)+1): REM SPIN THE WHEEL
210 REM LINES 210-300 DISPLAY THE WINNING NUMBER
220 PRINT "THE WINNER IS NUMBER "; X
300 REM: LINES 310-590 DETERMINE WINNINGS AND LOSSES
310 FOR J=1 TO N : REM FOR EACH PLAYER
320 IF X=B(J) THEN 400
325 GOTO 330
330 A(J)=A(J)-C(J): REM PLAYER J LOSES. DEDUCT BET
340 PRINT "PLAYER";J ;" LOSES"
350 GOTO 420
400 A(J)=A(J)+32*C(J): REM PLAYER J WINS. ADD WINNINGS
410 PRINT "PLAYER ";J;" WINS "; 32*C(J); " DOLLARS"
420 NEXT J
430 PRINT "PLAYER BANKROLLS"
440 PRINT
```

```
450 PRINT "PLAYER", "CHIPS"
460 FOR J=1 TO N
470 PRINT J,A(J)
480 NEXT J
500 INPUT "DO YOU WISH TO PLAY ANOTHER ROLL(Y/N)";R$
510 HOME
520 IF R$="Y" THEN 100
530 PRINT "THE CASINO IS CLOSED. SORRY!"
600 END
```

You should try a few spins of the wheel. The program is fun as well as instructive. Note that the program allows you to bet more chips than you have. In the exercises you'll be asked to add in a test that there are enough chips to cover the bet. You could also build lines of credit into the game!

Treat the output of the random number generator as you would any other number. In particular, you may perform arithmetic operations on the random numbers generated. For example, 5*RND(1) multiplies the output of the random number generator by 5, and RND(1) + 2 adds 2 to the output of the random number generator. Such arithmetic operations are useful in producing random numbers from intervals other than 0 to 1. For example, to generate random numbers between 2 and 3, use RND(1) + 2.

Example 3. Write a program which generates 10 random numbers lying in the interval from 5 to 8.

Solution. Let's build up the desired function in two steps. Start from the function RND(1), which generates numbers from 0 to 1. First, adjust for the length of the desired interval. From 5 to 8 is 3 units, so multiply RND(1) by 3. The function 3*RND(1) generates numbers from 0 to 3. Now adjust for the starting point of the desired interval, namely 5. By adding 5 to 3*RND(1), you obtain numbers lying between 0 + 5 and 3 + 5, that is, between 5 and 8. Thus, 3*RND(1) + 5 generates random numbers between 5 and 8. Here's the program required.

```
10 FOR J=1 TO 10
20 PRINT 3*RND(1)+5
30 NEXT J
40 END
```

Example 4. Write a function to generate random integers from among 5,6,7,8,...,12.

Solution. There are 8 consecutive integers possible. Start with the function 8*RND(1), which generates random numbers between 0 and 8. Since you wish your random number to begin with 5, add 5 to get 8*RND(1) + 5. This produces random numbers between 5.000000000 and 12.99999999. Now use the **INT** function to chop off the decimal part. This yields the desired function:

```
INT(8*RND(1)+5)
```

Exercises (answers on page 204)

Write Floating Point BASIC functions which generate random numbers of the following sorts.

1. Numbers from 0 to 100.
2. Numbers from 100 to 101.
3. Integers from 1 to 50.
4. Integers from 4 to 80.
5. Even integers from 2 to 50.
6. Numbers from 50 to 100.
7. Integers divisible by 3 from 3 to 27.
8. Integers from among 4,7,10,13,16,19, and 22.
9. Modify the dice program so that it keeps track of payoffs and bankrolls, much like the roulette program in Example 2 (page 74). Here are the payoffs on a bet of one dollar for the various bets:

outcome	payoff
2	35
3	17
4	11
5	8
6	6.20
7	5
8	6.20
9	8
10	11
11	17
12	35

10. Modify the roulette program of Example 2 to check that a player has enough chips to cover the bet.
11. Modify the roulette program of Example 2 to allow for a $100 line of credit for each player.
12. Construct a program which tests one-digit arithmetic facts, with the problems randomly chosen by the computer.
13. Make up a list of ten names. Write a program which will pick four of the names at random. (This is a way of impartially assigning a nasty task!)

ANSWERS TO TEST YOUR UNDERSTANDING 1,2, and 3

1: 30%

2:
```
10 FOR J=1 TO 50
20 PRINT INT(2*RND(1)+1)
30 NEXT J
40 END
```

```
3:  10 LET X=INT(3*RND(1)+1)
    20 IF X=1 THEN PRINT "HEADS"
    30 PRINT "TAILS"
    40 END
```

3.5 Subroutines

In writing programs it's often necessary to use the same sequence of instructions more than once. It may not be convenient (or even feasible) to retype the set of instructions each time it's needed. Fortunately, Floating Point BASIC offers a convenient alternative: the subroutine.

A **subroutine** is a program which is incorporated within another, larger program. The subroutine may be used any number of times by the larger program. Often, the lines corresponding to a subroutine are isolated toward the end of the larger program. This arrangement is illustrated in Figure 3-1. The arrow to the subroutine indicates the point in the larger program at which the subroutine is used. The arrow pointing away from the subroutine indicates that, after completion of the subroutine, execution of the main program resumes at the point at which it was interrupted.

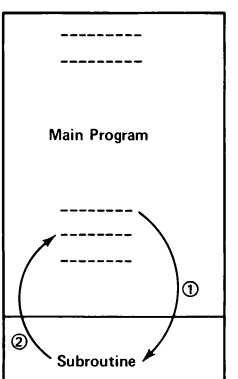


Figure 3-1. A subroutine.

Subroutines are handled with the pair of instructions **GOSUB** and RETURN. The statement

```
100 GOSUB 1000
```

sends the computer to the subroutine which begins at line 1000. The computer starts at line 1000 and carries out statements in order. When a RETURN statement is reached, the computer goes back to the main program, starting at the first line after 100. The next example illustrates the use of subroutines.

Example 1. Modify the roulette program of Example 2 (page 74), so that it allows bets on EVEN and ODD. A one dollar bet on either of these pays one dollar in winnings.

Solution. Your program will now allow three different bets: on a number and on EVEN or ODD. Design subroutines, corresponding to each of these bets, which determine whether player J wins or loses. For each subroutine, let X be the number (1-38) which results from spinning the wheel. In the preceding program, a bet by player J was described by two numbers: B(J) equals the number bet and C(J) equals the amount bet. Now let's add a third number to describe a bet. Let D(J) equal 1 if J bets on a number, 2 if J bets on EVEN, and 3 if J bets on ODD. In case D(J) is 2 or 3, again let C(J) equal the amount bet, but B(J) will be 0. The subroutine for determining the winners of bets on numbers can be obtained by making small modifications to the corresponding portion of our previous program, as follows:

```
1000 IF B(J)=X THEN 1100
1005 GOTO 1010
1010 PRINT "PLAYER ";J;" LOSES"
1020 A(J)=A(J)-C(J)
1030 RETURN
1100 PRINT "PLAYER ";J;" WINS";32*C(J);"DOLLARS"
1110 A(J)=A(J)+32*C(J)
1120 RETURN
```

Here's the subroutine corresponding to the bet EVEN.

```
2000 FOR K=1 TO 19
2010 IF X=2*K THEN 2100
2015 GOTO 2020
2020 NEXT K
2030 PRINT "PLAYER ";J;" LOSES"
2040 A(J)=A(J)-C(J)
2050 RETURN
2100 PRINT "PLAYER ";J;" WINS ";C(J);" DOLLARS"
2110 A(J)=A(J)+C(J)
2120 RETURN
```

Finally, here's the subroutine corresponding to the bet ODD.

```
3000 K=0
3005 IF X=2*K THEN 3030
3010 K=K+1
3015 IF K > 19 THEN 3100
3020 GOTO 3005
3030 PRINT "PLAYER ";J;" LOSES"
3040 A(J)=A(J)-C(J)
3050 RETURN
3100 PRINT "PLAYER ";J;" WINS ";C(J);" DOLLARS"
3110 A(J)=A(J)+C(J)
3120 RETURN
```

Now you're ready to assemble the subroutines together with the main portion of the program, which is almost the same as before. The only essential alteration is that you must now determine, for each player, which bet was placed.

```
10 INPUT "NUMBER OF PLAYERS ";N
20 DIM A(5),B(5),C(5): REM AT MOST 5 PLAYERS ALLOWED
25 REM LINES 30-60 ALLOW PLAYERS TO PURCHASE CHIPS
30 FOR J=1 TO N : REM FOR EACH OF THE PLAYERS
40 PRINT "PLAYER"; J
50 INPUT "HOW MANY CHIPS "; A(J)
60 NEXT J
100 PRINT "LADIES AND GENTLEMEN!"
105 PRINT "PLACE YOUR BETS PLEASE!"
110 FOR J=1 TO N : REM FOR EACH OF THE PLAYERS
120 PRINT "PLAYER"; J
121 PRINT "BET TYPE:1=NUMBER BET, 2=EVEN, 3=ODD"
122 INPUT "BET TYPE (1,2,OR 3) "; D(J)
123 IF D(J)=1 THEN 130
124 INPUT "AMOUNT "; C(J)
125 GOTO 140
130 INPUT "NUMBER,AMOUNT "; B(J),C(J):REM INPUT BET
140 NEXT J
200 X=INT(38*RND(1)+1): REM SPIN THE WHEEL
210 REM LINES 210-300 DISPLAY THE WINNING NUMBER
220 PRINT "THE WINNER IS NUMBER"; X
300 REM LINES 310-330 DETERMINE WINNINGS AND LOSSES
310 FOR J=1 TO N : REM FOR EACH PLAYER
320 IF D(J)=1 THEN GOSUB 1000
330 IF D(J)=2 THEN GOSUB 2000
335 IF D(J)=3 THEN GOSUB 3000
340 NEXT J
430 PRINT "PLAYER BANKROLLS"
440 PRINT
450 PRINT "PLAYER", "CHIPS"
460 FOR J=1 TO N
470 PRINT J,A(J)
```

```
480 NEXT J
500 INPUT "DO YOU WISH TO PLAY ANOTHER ROLL(Y/N)";R$
510 HOME
520 IF R$="Y" THEN 100
530 PRINT "THE CASINO IS CLOSED. SORRY!"
600 END
1000 IF B(J)=X THEN 1100
1005 GOTO 1010
1010 PRINT "PLAYER ";J;" LOSES"
1020 A(J)=A(J)-C(J)
1030 RETURN
1100 PRINT "PLAYER ";J;" WINS"; 32*C(J);" DOLLARS"
1110 A(J)=A(J)+32*C(J)
1120 RETURN
2000 FOR K=1 TO 19
2010 IF X=2*K THEN 2100
2015 GOTO 2020
2020 NEXT K
2030 PRINT "PLAYER ";J;" LOSES"
2040 A(J)=A(J)-C(J)
2050 RETURN
2100 PRINT "PLAYER ";J;" WINS ";C(J);" DOLLARS"
2110 A(J)=A(J)+C(J)
2120 RETURN
3000 K=1
3005 IF X=2*K THEN 3030
3010 K=K+1
3015 IF K > 19 THEN 3100
3020 GOTO 3005
3030 PRINT "PLAYER ";J;" LOSES"
3040 A(J)=A(J)-C(J)
3050 RETURN
3100 PRINT "PLAYER ";J;" WINS ";C(J);" DOLLARS"
3110 A(J)=A(J)+C(J)
3120 RETURN
3130 END
```

Note how the subroutines help organize your programming. Each subroutine is easy to write. Each is a small task and you'll have less to think about than when considering the entire program. It's advisable to break a long program into a number of subroutines. Not only is it easier to write in terms of subroutines, but it's much easier to check the program and to locate errors since subroutines may be individually tested.

TEST YOUR UNDERSTANDING 1 (answer on page 83)

Consider the following program.

```
10 GOSUB 500
20 A=5
500 B=7
510 GOSUB 600
600 C=9
700 RETURN
800 END
```

What is the line executed after line 700?

Example 2. Here are production figures for six assembly lines of a factory for January 1980 and January 1981. Calculate the percentage increase (or decrease) for each assembly line and determine the assembly line with the largest percentage increase.

Assembly Line	January 1980	January 1981
1	235,485	239,671
2	298,478	301,485
3	328,946	322,356
4	315,495	318,458
5	198,487	207,109
6	204,586	221,853

Solution. Plan the program in terms of subroutines. Store the January 1980 data in the array A(J) and the January 1981 data in the array B(J), J = 1,2,3,4,5,6. The first step will be to read the data into the arrays from **DATA** statements. The second step will be to calculate the percentage increase (decrease) for each assembly line. This is done by using a subroutine which you can start at line 1000. Store the percentage increases in the array C(J), J = 1,2,3,4,5,6. Finally, determine which of the numbers C(J) is the largest. This calculation can be carried out in a subroutine beginning in line 2000. Let K be the number of the assembly line with the largest percentage increase. You can then write your program as follows:

```
10 DIM A(6),B(6),C(6)
20 DATA  235485,  239671,  298478,  301485
30 DATA  328946,  322356,  315495,  318458
40 DATA  198487,  207109,  204586,  221853
50 FOR J=1 TO 6: REM READ ARRAYS
60 READ A(J), B(J)
70 NEXT J
80 FOR J=1 TO 6: REM COMPUTE PERCENT INCREASES
90 GOSUB 1000
```

```
100 NEXT J
200 PRINT "ASSEMBLY LINE","PERCENT INCREASE"
210 FOR J=1 TO 6: REM DISPLAY PERCENT INCREASES
220 PRINT J, C(J)
230 NEXT J
300 GOSUB 2000: REM DETERMINE LARGEST PERCENT INCREASE
310 PRINT "ASSEMBLY LINE ";K;" IS THE WINNER"
400 END
```

This program is not complete. It's still necessary to complete the subroutines in lines 1000 and 2000. The point to make here is that the use of subroutines allowed you to organize the program and to plan it so that programming may be done in small steps. It's now possible to write the subroutines without worrying about the program in its entirety. In order for you to obtain some practice, you'll be able to construct two subroutines for the exercises on page 82. (If you're impatient, you may look at the answers on page 206!)

Here's a useful variation of the **GOSUB** statement. Suppose that you wish to use:

1. the subroutine beginning at line 100 if the value of J is 1,
2. the subroutine beginning at line 500 if the value of J is 2, and
3. the subroutine beginning at line 1000 if the value of J is 3.

This complex set of decisions can be requested by a simple statement of the form

```
10 ON J GOSUB 100,500,1000
```

If the value of J is 1, the program goes to line 100; if the value of J is 2, the program goes to line 500; if the value of J is 3, the program goes to line 1000. If the value of J is not an integer, then any fractional part will be ignored. For example, if the value of J is 5.5, then the value 5 will be used. After dropping the fractional part, the value must be between 0 and 255 or an error will occur. If the value of J does not correspond to a given subroutine (that is, if J is 0 or more than 3 in the above example), then the instruction will be ignored. Instead of J, you could use any valid expression, such as J^2 + 3 or 3*J - 2. The expression will be evaluated and all of the above rules will apply.

Exercises (answers on page 206)

1. Write a subroutine computing the value of 5*J^2 - 3*J. Incorporate it in a program evaluating the given expression at J = .1, .2, .3, .4, .5.
2. Write the subroutine of Example 2 on page 81 which computes the percentage rates of change for each assembly line. Use the formula

```
<% rate of change>
 = 100 X (<Jan.1981 Prod.>-<Jan.1980 Prod.>)/<Jan.1980
Prod.>
```

3. Write a subroutine for Example 2 determining the largest percentage increase. Hint: Let the variable M hold the largest number among C(1),...,C(6). At the start, set M equal to C(1). Repeatedly compare M to C(2),C(3),...,C(6). After each comparison, replace M by the larger of M and what it's compared to. At the end of 5 comparisons, M will contain the largest among C(1),C(2),...,C(6). You may then determine which assembly line that value of M belongs to. That is, is the final value of M equal to C(1), C(2), C(3), C(4), C(5), or C(6)? For example, if M equals C(5), then assembly line 5 has the largest percentage increase.
4. Run the program of Example 2 with the subroutines in place.
5. Modify the roulette program of Example 1 (page 78) to allow the following bets: first 12 (1–12), second 12 (13–24), and third 12 (25–36). A winning one dollar bet of this type pays two dollars. In adding these bets, you may find the **ON ... GOSUB** instruction helpful.

ANSWER TO TEST YOUR UNDERSTANDING 1

1: 600

4

Easing Programming Frustrations

As you've probably discovered by now, programming can be a tricky and frustrating business. You must first figure out the instructions to give the computer. Next, you must type the instructions into RAM. Finally, you must run the program and, usually on the first run, figure out why your program won't work. This process can be tedious and frustrating, especially in dealing with long or complex programs. Realize, however, that programming frustrations often result from the limitations and inflexibility of the computer to understand exactly what you're saying. In talking with another person, you usually sift out irrelevant information, correct minor errors, and still maintain the flow of communication. With a computer, however, you must first clear up all of the imprecisions before the conversation can even begin.

Fortunately, there are ways to ease programming burdens to track down errors and correct them. These features should help you develop programs quicker and with fewer errors.

4.1 Editing Program Lines

Suppose that you discover a program line with an error in it. How can you correct it? Up to now, the only way was to retype the line. There is a much better way. The Franklin allows you to **edit** lines. That is, you may add, delete, or change text in existing program lines.

On the Franklin, the editing process consists of four steps:

1. Display the line on the screen.
2. "Copy" the line by using the forward arrow key to move the cursor over each character of the line.

3. Add, change, or delete characters as they arise in the "copy" process.
4. Send the line to the computer via the [RETURN] key.

The best way to understand the editing process is to work through several examples by typing them out on your keyboard.

Suppose you've typed the following program lines:

```
10 PRIMT X,Y,Z
20 IF A=5 THN 50: GOTO 30
]■
```

The third line indicates the cursor position. You can see immediately that there are two spelling errors, PRIMT and THN. (If the computer had any common sense, it would have known what you meant.) In addition, suppose that you want to change X,Y,and Z in the first line to read: A,X,Y,Z. Finally, suppose you wish to delete :GOTO 30 on the second line. Let's use the editing process to correct them.

The fundamental idea of editing on the Franklin is to recopy the line while making the needed corrections. The recopy and correction procedures require that you move the cursor by using certain key combinations. Let's begin by describing how to move the cursor.

The following keys move the cursor without affecting anything already on the screen.

I—move cursor one space up
J—move cursor one space to the left
K—move cursor one space right
M—move cursor one space down

Of course, these keys also generate the letters I,J,K, and M. To use them for moving the cursor, make sure you're in upper case and then hit the key marked [ESC] (ESCAPE). You may return these keys to their normal functions by typing any key other than I,J,K,M, or [ESC]. In moving the cursor, you may find it helpful to hold the key down. If you do this while pressing I, for instance, the cursor will execute a continuous move up. You may stop the cursor by lifting your fingers from the keys.

Let's get back to our lines to be corrected. The lines are already displayed on the screen (Step 1). Next, move the cursor to the first digit of the line number of the first line to be corrected. You copy any characters which are correct by moving the cursor across them using the forward arrow key. (Note that the cursor motion key K will not do for this purpose!) Move the cursor until it rests on the M in PRIMT . To correct the M, type N. The display now looks like this:

```
10 PRINT X,Y,Z
20 IF A=5 THN 50: GOTO 30
```

The first error has now been corrected. Now continue to copy the line with the forward arrow key. Move across the T and the space and put the cursor on the X. You must insert the characters **A** and , before the X. Here's the display:

```
10 PRINT X,Y,Z
20 IF A=5 THN 50 :GOTO 30
```

To insert text at the cursor position, type ESC I to move the cursor up one line. Then type the material to be inserted. Here's the display:

```
      A,_
10 PRINT X,Y,Z
20 IF A=5 THN 50: GOTO 30
```

Now type ESC M to bring the cursor down one line and type J three times to move the cursor back to the X. Now resume the copy operation using the forward arrow key. After you pass over the last character of the line, hit RETURN. This finishes the corrections on the first line.

As the above example shows, it's a good idea to have a blank line above a line in which you wish to make corrections. Line 20 is on the screen, but has line 10 above it. At this point, you should type `LIST 20` followed by RETURN. This will display line 20 with a blank line above it.

```
LIST 20

20 IF A=5 THN 50 :GOTO 30
]■
```

Move the cursor to the first character of line 20 and begin the copy operation as you did in correcting line 10. Correct the misspelling of **THEN** by moving the cursor to the N (14 spaces to the right), and proceeding just as you did in correcting the **A, X, Y** error. The final correction is to delete `:GOTO 30`. This is done by positioning the cursor on the `0` and hitting the backspace key ←. As you backspace over a letter, it's erased. Finally, hit the RETURN key to enter the corrected line. Here's an easier way of deleting the **:GOTO 30.** Position the cursor at the first character to be deleted (the :) and hit the RETURN key.

The above example illustrates the various editing features of the Franklin. You may use the editing keys in the same way to alter any line on the screen. If you wish to alter a program line which is not currently on the screen, you may display the desired line using the **LIST** command. Editing would then take place as shown.

IMPORTANT NOTE: Editing changes occur only in the copy of the program in RAM. In order for changes to be reflected in copies of the program on diskette, it's necessary to save the edited copy of the program.

Exercises

What keystrokes accomplish the following editing functions?

1. Move the cursor four spaces to the right.
2. Delete the fourth letter to the right of the cursor.
3. Insert the characters 538 at the current cursor position.
4. Delete the portion of the line to the right of the cursor position.
5. Move the cursor up eight spaces.
6. Move the cursor to the right three spaces.
7. List the current version of the line.
8. Change 0 to a 1 at the current cursor position.
9. Delete the letter "a" eight spaces to the left of the current cursor position.
10. Cancel all changes in the current line.

Use the line editor to make the indicated changes in the following program line. The exercises are to be done in order.

```
300 FOR M=11 TO 99, SETP .5 : X=M^2 -5
```

11. Delete the ,
12. Correct the misspelling of the word STEP.
13. Change M^2-5 to M^3-2
14. Change .5 to -1.5
15. Add the following characters to the end of the line: Y = M + 1.

4.2 Flow Charting

In the last two chapters, the programs were fairly simple. By the end of Chapter Three they were becoming more involved. However, there are many programs which are much more lengthy and complex. You might be wondering how it's possible to plan and execute such programs. The key idea is to reduce large programs to a sequence of smaller programs which can be written and tested separately.

The old saying "A picture is worth a thousand words" is true for computer programming. In designing a program, especially a long one, it's helpful to draw a picture depicting the instructions of the program and their interrelationships. Such a picture is called a flowchart.

A flowchart is a series of boxes connected by arrows. Within each box is a series of one of more computer instructions. The arrows indicate the logical flow of the instructions. For example, the following flowchart shows a program for calculating the sum $1 + 2 + 3 + \ldots + 100$.

The arrows indicate the sequence of operations. Note that between the second and third box there is a notation stating J = 1,2,...,100. This notation indicates a loop on the variable J. This means the operation in the third box is to be

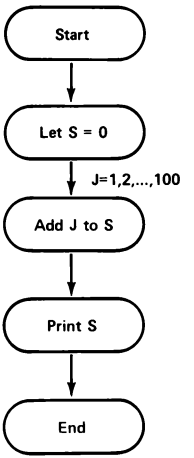


Figure 4-1.

repeated 100 times—for J = 1,2,...,100. Note how easy it is to proceed from the above flowchart to the corresponding BASIC program:

```
10 LET S=0                    (box 2)
20 FOR J=1 TO 100
30 LET S=S+J                  (box 3)
40 NEXT J
50 PRINT S                    (box 4)
60 END                        (box 5)
```

There are many flow charting rules. Different shapes of boxes represent certain programming operations. Let's adopt a very simple rule that all boxes are rectangular, except for decision boxes. Decision boxes are diamond-shaped. The following flowchart shows a program which decides whether a credit limit has been exceeded.

Note that the diamond-shaped block contains the decision "Is D > Limit (L)?" Corresponding to the two possible answers to the question, there are two arrows leading from the decision box. Note also how you use the various boxes to help assign letters to the program variables. Once the flowchart is written, it's easy to transform it into the following program:

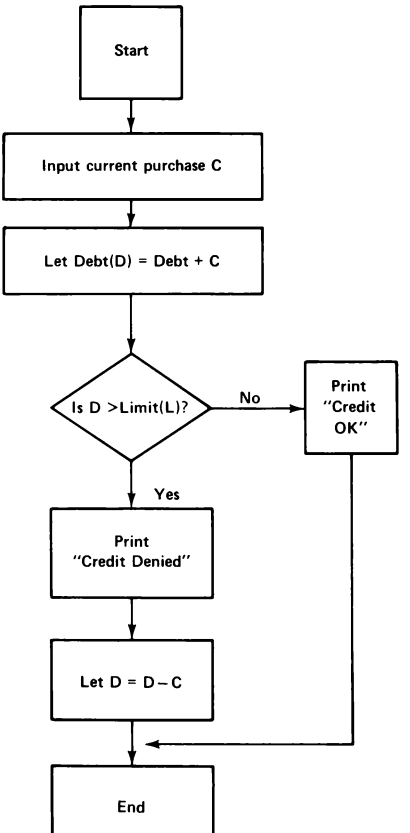

Start
Input current purchase C
Let Debt(D) = Debt + C
Is D >Limit(L)?
No
Print "Credit OK"
Yes
Print "Credit Denied"
Let D = D−C
End

Figure 4-2.

```
10 INPUT C    (boxes 1,2)
20 INPUT D,L
30 LET D=D+C  (box 3)
40 IF D > L THEN 100  (box 4)
50 GOTO 200 ("Yes" arrow)
100 PRINT "CREDIT DENIED" (box 6)
110 LET D=D-C  (box 7)
120 GOTO 300
200 PRINT "CREDIT OK" (box 5)
300 END  (box 8)
```

You'll find flow charting helpful in thinking out the necessary steps of a program. As you practice flowcharting, you'll develop your own style and conventions. That's fine, as long as they help you write programs.

Exercises (answers on page 207)

Draw a flowchart planning computer programs to do the following.

1. Calculate the sum $1^2+2^2+\ldots+100^2$, print the result, and determine whether the result is larger than, smaller than, or equal to 487^3.
2. Calculate the time elapsed since the computer was turned on.
3. The roulette program of Section 3.5 (see page 78).
4. The payroll program in Example 2 of Section 3.2 (see page 62).

4.3 Errors and Debugging

An error is sometimes called a "bug" in computer jargon. The process of finding these errors or "bugs" in a program is called **debugging.** This can often be a ticklish task. Manufacturers of commercial software must regularly repair bugs users discover in their programs! Your Franklin is equipped with a number of features to help detect bugs.

The Trace

Often your first try at running a program results in failure, while giving you no indication as to why the program is not running correctly. For example, your program might just run indefinitely, without giving you a clue as to what it is actually doing. How can you figure out what's wrong? One method is to use the **trace** feature. Let's illustrate use of the trace by debugging the following program designed to calculate the sum $1+2+\ldots+100$.

```
10 LET S=0
20 LET J=0
30 LET S=S+J
40 IF J=100 THEN 100
```

```
50 GOTO 200
100 LET J=J+1
110 GOTO 20
200 PRINT S
300 END
```

This program has two errors in it. (Can you spot them right off?) All you know initially is that the program is not functioning normally. The program runs, but prints out the answer 0, which you recognize as nonsense. How can you locate the errors? Let's turn on the trace function by typing **TRACE.** The computer will respond with] ■. Now type **RUN.** The computer will run the program and print out the line numbers of all executed instructions. Here is what the display looks like:

```
TRACE
]■

RUN
#10 #20 #30 #40 #200 0
#300
```

The numbers preceded by # indicate the line numbers executed. That is, the computer executes, in order, lines 10, 20, 30, 40, 200, and 300. The 0 not preceded by # is the program output resulting from the execution of line 200. The list of line numbers is not what we were expecting. The program was designed (or so you thought) to execute line 100 after line 40. No looping is taking place. How did you get to line 200 after line 40? This suggests that you examine line 40: Lo and behold! There's an error. The line numbers 100 and 200 appearing in lines 40 and 50 have been interchanged (an easy enough mistake to make). Let's correct this error by retyping the lines 40 and 50:

```
40 IF J=100 THEN 200
50 GOTO 100
```

In triumph, you run our program again. Here's the output:

```
#10 #20 #30 #40 #100 #110 #20 #30
#40 #100 #110 #20 #30 #40 #100 #110
#20 #30 #40 #100
BREAK IN 110
```

Actually, the above output goes whizzing by as the computer races madly on executing the instructions. After about 30 seconds, you sense that something is indeed wrong since it's unlikely that your program could take this long. Stop execution by means of the key combination CTRL C. The last line indicates that you interrupted the computer while it was executing line 110. Actually, your screen will be filled with output resembling the above. You'll notice that the computer is in a loop. Each time it reaches line 110, the loop goes back to line 20. Why doesn't the loop ever end? In order for the loop to terminate, J must equal 100. Well, can J ever equal 100? Of course not! Every time the computer executes line 20, the value of J is reset to 0. Thus, J is never equal to

100 and line 40 always sends you back to line 20. You clearly don't want to reset J to 0 all the time. After increasing J by 1 (line 100), you wish to add the new J to S. You want to go to 30, not 20. Correct line 110 to read

```
110 GOTO 30
```

RUN the program again. There'll be a rush of line numbers on the screen followed by the output 5050, which appears to be correct. Your program is now running properly. Turn off the trace by typing **NOTRACE** (Trace off). Finally **RUN** the program once more for good measure. The above sequence of operations is summarized in the following display:

```
#40 #200 5050
#300
]■

NOTRACE
]■

RUN
5050
]■
```

Note that the trace function cannot be used when using disk operations discussed later in this book.

Error Messages

In the example above, the program actually ran. A more likely occurrence is that there is a program line (or lines) which the computer is unable to understand due to an error or some other sort of problem. In this case, program execution ends too soon. The computer can often help in this instance since it's trained to recognize many of the most common errors. The computer will print an error message indicating the error type and the line number in which it occurred. You should immediately **LIST** the indicated line and attempt to find the cause of the error. Suppose that the error reads:

```
SYNTAX ERROR IN 530
```

To analyze the error, you type

```
LIST 530
```

resulting in the display

```
530 LET Y=(X+2(X^2-2)
```

Note that there is an open parenthesis (without a corresponding close parenthesis). This is enough to trigger an error so modify line 530 to read

```
530 LET Y=X+2(X^2-2)
```

RUN the program again and you find that there is still a syntax error in line 530! This is the frustrating part since not all errors are easy to spot. However, if you look closely at the expression on the right, you'll note that there's no * to indicate the product of 2 and (X^2–2). This is a common mistake, especially for those familiar with the use of algebra. (In algebra, the product is usually indicated without any operation sign.) Correct line 530 again. (You may either retype the line or use the line editor.)

```
530 LET Y=X+2*(X^2-2)
```

Now you find that there is no longer a syntax error in line 530!

Exercises (answers on page 209)

1. Use the error messages to debug the following program to calculate $(1^2 + 2^2 + \ldots + 50^2)(1^3 + 2^3 + \ldots + 20^3)$.

```
10 LET S="0"
20 FOR J=1 TO 100
30 S=S+J(2
40 NEXT K
50 LET T=0
60 FOR J=1 TO 30
70 LET T=T +J^3
80 NXT T
90 NEXT T
100 LET A=ST
110 PRINT THE ANSWER IS, A
120 END
```

2. Use the trace function to debug the following program to determine the smallest integer N for which N^2 is larger than 175263.

```
10 LET N=0
20 IF N^2 > 175263 THEN 100
30 PRINT "THE FIRST N EQUALS"
100 N=N+1
110 GOTO 10
200 END
```

4.4 APPENDIX—Some Common Error Messages

? SYNTAX ERROR—There is an unclear instruction (misspelled?), mismatched parentheses, incorrect punctuation, illegal character, or illegal variable name in the program.

? UNDEF'D STATEMENT ERROR—The program uses a line number which does not correspond to an instruction. This can easily occur from deleting lines which are mentioned elsewhere. It can also occur when testing a portion of a program which refers to a line not yet written.

? UNDEF'D FUNCTION ERROR—The program uses a function which has not been defined in a **DEF** statement.

? OVERFLOW ERROR—A number too large for the computer.

? DIVISION BY ZERO ERROR—Attempting to divide by 0. This may be a hard error to spot. The computer will round to 0 any number smaller than the minimum allowed. Use of such a number in subsequent calculations could result in division by 0.

? ILLEGAL QUANTITY ERROR—(For the mathematically-minded.) Attempting to evaluate a function outside of its mathematically defined range. For example, the square root function is defined only for non-negative numbers, the logarithm function only for positive numbers, and the arctangent only for numbers between -1 and 1. Any attempt to evaluate a function at a value outside these respective ranges will result in an illegal quantity error.

? BAD SUBSCRIPT ERROR—Attempting to use an array with one or more subscripts outside the range allowed by the appropriate **DIM** statement.

? STRING TOO LONG ERROR—Attempting to specify a string containing more than 255 characters.

? OUT OF MEMORY ERROR—Your program will not fit into the computer's memory. This could result from large arrays, too many program steps, or a combination of the two.

? FORMULA TOO COMPLEX ERROR—Due to the internal processing of your formula, your formula resulted in a string expression that was too long or complex. This error can be corrected by breaking the string expression into a series of simpler expressions.

? TYPE MISMATCH ERROR—Attempting to assign a string constant as the value of a numeric variable, or a numeric constant to a string variable.

? REDIM'D ARRAY ERROR—Attempting to **DIM** an array which has already been dimensioned. Note that once you refer to an array within a program, even if you don't specify the dimensions, the computer will regard it as being dimensioned at 10.

? NEXT WITHOUT FOR ERROR—A **NEXT** statement which does not correspond to a **FOR** statement.

? RETURN WITHOUT GOSUB ERROR—A **RETURN** statement encountered while not in a subroutine.

? OUT OF DATA ERROR—Attempting to read data which isn't there. This can occur in reading data from **DATA** statements, cassettes, or diskettes.

? CAN'T CONTINUE ERROR—Attempting to give a **CONT** command after the program has ENDed, or before the program has been **RUN** (such as after an edit session).

5

Your Computer As A File Cabinet

In the previous chapters, you learned the fundamentals for programming the Franklin, without any direct discussion of disk files. This chapter discusses the operation of these devices and their application to writing and reading programs and data files. Section 5.1 discusses the difference between program files and data files. Sections 5.2, 5.3, and 5.4 provide an introduction to disk drive operations.

5.1 What Are Data Files?

Computer programs as used in business and industry usually refer to files of information which are stored within the computer. For example, a personnel department would keep a file of personal data on each employee, containing name, age, address, social security number, date employed, position, salary, and so forth. A warehouse would maintain an inventory for each product, with the following information: product name, supplier, current inventory, units sold in the last reporting period, date of last shipment, size of last shipment, and units sold in the last 12 months. Files like this are called **data files.** They could contain hundreds, thousands, or even hundreds of thousands of entries.

Data files are usually stored on a mass storage device. In the case of the Franklin, this storage device is a disk drive. Data files are to be distinguished from **program files,** which result from saving programs, even though program files and data files exist side by side in mass storage.

The following example will give you a better idea of how a file is organized within mass storage. Suppose that a teacher stores grades in a data file. For each student in the class, there are four exam grades. A typical entry in the data file would contain the following data items:

```
student name, exam grade #1, exam grade #2, exam grade #3,
exam grade #4
```

In a data file, the data items are organized in sequence. So, the beginning of the above data file might look like this:

```
"John Smith", 98, 87, 93, 76, "Mary Young", 99, 78, 87, 91,
"Sally Ronson", 48, 63, 72, 80, . . .
```

The data file consists of a sequence of either string constants (the names) or numeric constants (the grades), with the various data items arranged in a particular pattern (name followed by four grades). This particular arrangement is designed so the file may be read and understood. For instance, if you read the data items above, you know in advance that the data items are in groups of five with the first one a name and the next four the corresponding grades. In order for the computer to know where one data item ends and another begins, you must separate consecutive data items by characters called **delimiters.** Some examples of delimeters are: spaces, commas, and RETURN.

Your Franklin computer can read data from a data file while a program is running, so the program can use the data. The file may be used within the program. For example, a personnel department might have a program which (1) enters changes in the personnel data file and (2) displays requested information about a given employee. To perform task (1), the program would read the personnel data file, alter the relevant items and rewrite the file into mass storage. To perform task (2), the program would read the data file, search for the requested information and display it on the screen or printer. In effect, your computer is serving as a convenient file cabinet for the storage of data. Moreover, the programming capability of the computer allows you to easily "shuffle through" the data for a specific piece of information.

5.2 Using Franklin Diskette Files

This chapter discusses the mechanics of setting up, writing, and reading data files. It will also introduce the operation and use of diskette files.

On Diskettes and Diskette Files

The Franklin lets you install one or more disk drives. Each disk drive must be connected to a disk controller card which is inserted in one of the empty slots in the main system unit. A single controller card may control one or two disk drives. A particular disk drive is described by two numbers: the slot number in which its controller card is inserted and the number 1 or 2, indicating the disk drive on the particular controller card. The slot number is indicated in commands by the letter S and the drive number is indicated by the letter D. Thus, for example,

```
S3, D2
```

refers to disk drive 2 attached to the controller card in slot 3.

To store information, the disk drives use 5 1/4-inch floppy diskettes. Each diskette can accommodate approximately 143,000 characters (about 50 double-spaced typed pages). Figure 5-1 illustrates the essential parts of a diskette. The jacket is designed to protect the diskette. The interior of the jacket contains a lubricant which helps the diskette rotate freely within the jacket. The diskette is sealed inside and you should never attempt to open the protective jacket.

The disk drive reads and writes on the diskette through the **read-write window.** As you probably already know, diskettes are very fragile. Never, under any circumstances, touch the surface of the diskette. A small piece of dust or even oil from a fingerprint can damage the diskette and render all the information on it totally useless.

The **write protect notch** allows you to prevent changes to information on the diskette. When this notch is covered, with one of the metallic labels provided with the diskettes, the computer may read the diskette, but it won't write or change any information on the diskette. To write on a diskette, you must uncover the write protect notch.

Starting the Computer

The version of Floating Point BASIC contained in ROM is not powerful enough to control the flow of information to and from the disk drives. For this purpose, you need a program called the **operating system** which acts as a manager for all the activities which go on in the computer, coordinating the flow of information between the keyboard, video display, RAM, ROM, disk drives, and any peripheral devices which you may have added to your computer system. Franklin provides you with an operating system called **DOS.** This program is contained on a **DOS diskette** which is included when you purchase your disk drives.

In order to make use of your disk drives, you've got to read a backup copy of **DOS** into the computer. To do so, follow this procedure:

1. Insert the **DOS** diskette into drive 1.
2. Close the door.
3. Turn on the computer to boot the diskette.
4. You'll see the usual title and version number.

TEST YOUR UNDERSTANDING 1

a. Start your disk operating system.
b. Type in and run the following program

```
10 PRINT 1+3+5+7
20 END
```

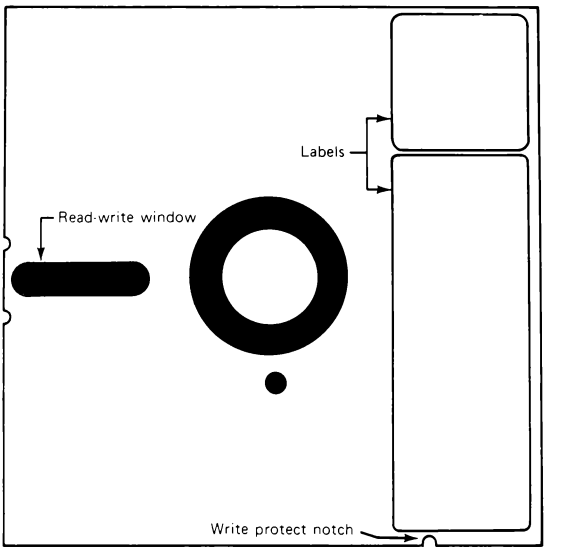


Figure 5-1. A diskette.

Using Your Disk System

As you remember from *The Franklin User Reference Manual,* blank diskettes are so empty that the computer can't use them to store programs or data until it puts a sort of organizational plan on them.

This way the computer can easily find out what files are on the diskette, where they're located, and where there's unused space. Putting this organizational scheme on a diskette is called **initializing** or **formatting.**

When you do it, you'll also be putting a version of **DOS,** the disk operating system, on the diskette. There's an advantage to this. No matter what you store on it, you'll be able to boot the diskette and use **DOS** commands any time without involving the **DOS** diskette.

You should know a few more things before you try the **INIT** command. The version of **DOS** that **INIT** puts on the blank diskette is generally called a "slave" **DOS.** Your diskette will work fine on your computer or on any other Franklin or Franklin-compatible machine that has exactly the same memory size as yours. But if you intend to use the diskette on another machine with a different memory capacity, you'll have to do one more thing to the diskette—make it a "master" using **FUD's M** command.

To initialize a diskette, take the **DOS** diskette out of drive 1 and insert a new diskette. Make sure that you're using a blank diskette or one that you don't need when you try this **INIT** example. If you forget and leave the **DOS** diskette in, you'll erase all of the programs and data on it.

Now type:

```
INIT HELLO
```

The Franklin will begin initializing the diskette in drive 1. You'll know it's done when you get the prompt on the next line. The diskette is now initialized, but you're not quite done yet.

When you initialize a diskette you can specify that any type A program be run automatically when the diskette is booted. Once you start using your programs, you'll have type A programs that you'll want to run automatically. This saves you the time and typing you'd have to spend calling upon the program and getting it to run. The **DOS** diskette, for example, was initialized with the **HELLO** program so that every time you boot it, the **HELLO** program automatically displays:

```
              ACE XXXX
            DOS DISKETTE

  Please wait a few seconds while I
         load INTEGER BASIC.

  Okay, everything is ready to go!
 You are now in Floating Point Basic.
```

You haven't gotten your new diskette to this stage yet. Even though you've just typed **INIT HELLO,** the **HELLO** program won't run and display the version messages when you boot your new diskette. You'll only see a prompt or two and a blinking cursor. You have an operating system on there and the word **HELLO,** but **HELLO** won't run. The program itself isn't on the diskette yet.

Put the **DOS** diskette in drive 1 and type:

```
LOAD HELLO
```

The **HELLO** program will go from the diskette into the computer's memory. Remove the **DOS** diskette and put the new diskette back in drive 1. Type:

```
SAVE HELLO
```

The **HELLO** program is now on the new diskette. Boot it and you'll see the usual version message (with one modification that isn't particularly relevant for a storage diskette).

Now you have a diskette you can use to store programs.

5.3 An Introduction to DOS

DOS uses the version of Floating Point BASIC stored in ROM, **DOS** also provides a number of additional commands which make Floating Point BASIC more flexible and easy to use. It is, however, beyond the scope of this book to cover all the special features of **DOS.** Still, here are a few of the most useful features of **DOS.**

Saving and Loading Programs

To save a program, use the **SAVE** command. For example, to save a program named **BUDGET** on the diskette in drive 1, use the command

```
SAVE BUDGET, D1
```

Here's another example. To save a program named **ROULETTE.011** on drive 2, use the command

```
SAVE ROULETTE.011, D2
```

Suppose that, at some later time, the diskette containing this program is moved to drive 1. The program may then be loaded into RAM via the command

```
LOAD ROULETTE.011, D1
```

If you omit the drive number, the computer will automatically assume that drive 1 is intended. Moreover, if you have only one disk drive, then you may omit the drive number in all file commands.

TEST YOUR UNDERSTANDING 1

a. Save the program

```
10 PRINT 5+7
20 END
```

under the name **BUTTER.**

b. Try to load the program without giving the correct name.

c. Load the program from diskette.

The Current Disk Drive

One of the disk drives is designated the **current disk drive.** If you give a **DOS** command without specifying which disk drive, **DOS** will automatically assume that you're referring to the current disk drive. As soon as you give a **DOS** command specifying a drive other than the current drive, the current drive is changed to the drive specified.

Suppose, for example, that the current disk drive is 1 and that you give the command:

```
SAVE INVENTORY
```

The program **INVENTORY** will then be saved on drive 1. If you then give a **DOS** command which involves drive 2, the current drive will change to drive 2.

DOS commands may be used in two ways. First, provided you've booted **DOS,** you can type a **DOS** command any time the Floating Point prompt is displayed. In this case, the computer is not executing a program. A **DOS** command given in this situation is executed immediately. When the command is executed, the Floating Point BASIC prompt is redisplayed.

A **DOS** command may also be used within BASIC programs. In this case, you use an instruction of the form:

```
10 PRINT CHR$(4); " DOS command"
```

The `CHR$(4)` is a control character which tells the computer that what follows is a **DOS** command. The actual **DOS** command is typed in quotation marks as shown above.

To give you an idea of what **DOS** can do, let's review some of the available **DOS** commands mentioned in *The User Reference Manual.* As long as you've booted **DOS** into the Franklin's memory, you can use **DOS** commands along with Floating Point BASIC.

Catalog

The **DOS** command **CATALOG** allows you to ask for a listing of the names of all files on a given diskette. For example, to display the directory of the diskette in drive 1, just type

```
CATALOG,D1
```

followed by RETURN. (You may omit the D1 if drive 1 is the current drive.)

Erasing Files

You may erase files from a diskette. For example, to erase the file **ROULETTE** type:

```
DELETE ROULETTE
```

Renaming a File

You may rename a file by using the **RENAME** command. For example, to change the name of **ROULETTE** to **GAME**, use the command

```
RENAME ROULETTE,GAME
```

Note that the old name always comes **first,** followed by the new name.

Exercises (answers on page 209)

1. a. Write a program which computes $1^2 + 2^2 + \ldots + 50^2$.
   b. **SAVE** the program under the name **SQUARES.**
   c. Type **ERASE RAM**
2. Recover the program of 1a. without retyping it using the **LOAD** command.
3. Erase the program **SQUARES** of 1a.
4. Make a copy of the diskette you're now using.

5.4 Data Files For Disk Users

In this section, you'll find the procedures for reading and writing data files on diskettes. In order to either read or write a data file on diskette, it's necessary to **OPEN** the file.

But before you see how to do that, it might be worthwhile to review a few conventions. File names can't be more than thirty characters long. The first character must be a letter, and none of the characters in the name can be commas, because they're used to separate file names from other things such as disk drive numbers. The commas you see in the examples must appear in the command exactly as they're shown. When **D1** and **D2** appear, they must be typed exactly as they're shown too, because they tell **DOS** which drive to go to in order to perform the command.

When opening a file, you tell **DOS** whether you plan to read the file or write the file. For example, to write a file with the name **INVOICE.034** on disk drive 2, you would use a sequence of instructions like this:

```
100 PRINT CHR$(4); "OPEN INVOICE.034, D2"
110 PRINT CHR$(4); "WRITE INVOICE.034, D2"
```

Note that these instructions do not actually write any data into the file. They merely prepare the file for later writing.

Suppose now that you wish to enter the following data into the file:

```
DJ SALES   $358.79   4/5/81
```

You would use the following instruction:

```
200 PRINT "DJ SALES", "$358.79", "4/5/81"
```

Once the **WRITE** command has been given, all subsequent **PRINT** statements will cause data to be written into the file indicated.

When you're finished writing a file, you must close it with a **CLOSE** instruction. To close the file you've just been considering, use the instruction

```
300 PRINT CHR$(4); "CLOSE INVOICE.034,D1"
```

Example 1. Create a data file consisting of names, addresses, and telephone numbers from your personal telephone directory. Assume that you'll type the addresses into the computer and tell the computer when the last address has been typed.

Solution. Use **INPUT** statements to enter the various data. Let A$ denote the name of the person, B$ the street address, C$ the city, D$ the state, E$ the zip code, and F$ the telephone number. For each entry, there's an **INPUT** statement corresponding to each of these variables. The program then writes the data to the diskette. Here's the program:

```
10 INPUT "NUMBER OF ENTRIES IN DIRECTORY";N
20 DIM A$(100),B$(100),C$(100),D$(100),E$(100),F$(100)
30 FOR J=1 TO N
110 INPUT "NAME "; A$(J)
120 INPUT "STREET ADDRESS "; B$(J)
130 INPUT "CITY "; C$(J)
140 INPUT "STATE "; D$(J)
150 INPUT "ZIP CODE "; E$(J)
160 INPUT "TELEPHONE "; F$(J)
170 NEXT J
200 PRINT CHR$(4); " OPEN TELEPHONE"
210 PRINT CHR$(4); "WRITE TELEPHONE"
220 FOR J=1 TO N
230 PRINT A$(J):?B$(J):?C$(J):?D$(J):?E$(J):?F$(J)
240 NEXT J
300 PRINT "END":?"":?"":?"":?""
310 PRINT CHR$(4); "CLOSE TELEPHONE"
320 END
```

The above program may be used to set up a directory consisting of up to 100 entries. You should set up such a computerized telephone directory of your own. It's very instructive. Moreover, when coupled with the search program

given below, it'll allow you to look up addresses and phone numbers using your computer.

TEST YOUR UNDERSTANDING 1

Use the above program to enter the following address into the file.

John Jones
1 South Main St., Apt. 308
Phila. Pa. 19107
527-1211

The **WRITE** instruction begins to write the file from the beginning. If you wish to add to an existing file, you should also use the **APPEND** instruction:

```
10 PRINT CHR$(4); "APPEND TELEPHONE"
20 PRINT CHR$(4); "WRITE TELEPHONE"
```

Such a sequence of instructions will cause any subsequent **PRINT** statements to print their data at the end of the file.

TEST YOUR UNDERSTANDING 2

Add to the telephone file started in TEST YOUR UNDERSTANDING 1 above the following entry.

Mary Bell
2510 9th St.
Phila. Pa. 19138
937-4896

WARNING: If you wish to replace a file which already exists with one of the same name, you should erase the first file using an instruction of the form

```
20 PRINT CHR$(4); "DELETE TELEPHONE"
```

Then you may safely write the new version using the instructions shown above. If you fail to delete the old version, you run the risk of mixing the old and new versions. In short, you may create a ghastly mess!

Let's now discuss the procedure for reading data files from a diskette. As is the case with writing files, it's necessary to open the file first. Consider the telephone file in Example 1 (page 105). To open it for input, you could use the instruction

```
300 PRINT CHR$(4); "OPEN TELEPHONE"
310 PRINT CHR$(4); "READ TELEPHONE"
```

Once the file is open, it may be read via the instruction

```
400 INPUT A$,B$,C$,D$,E$,F$
```

This instruction will read one of the telephone-address entries from the file. In order to read a file, it's necessary to know the precise format of the data in the file. For example, the form of the above **INPUT** statement was dictated by the fact that each telephone-address entry was entered into the file as six consecutive string constants, separated by commas. The input statement works like any other input statement: Faced with a list of items separated by commas, it assigns values to the indicated variables, in the order in which the data items are presented. Note that the commas in the data file are essential. In order for an **INPUT** statement to assign values to several variables at once, the values must be separated by commas!

Example 2. Write a program which searches for a particular entry of the telephone directory file created in Example 1.

Solution. **INPUT** the name corresponding to the desired entry. The program will then read the file entries until a match of names occurs. Here's the program:

```
10 INPUT "NAME TO SEARCH FOR "; Z$
15 PRINT CHR$(4); "OPEN TELEPHONE"
16 PRINT CHR$(4); "READ TELEPHONE"
20 INPUT A$,B$,C$,D$,E$,F$
30 IF A$=Z$ THEN 100
40 IF A$="END" THEN 200
50 GOTO 20
100 HOME
110 PRINT A$
120 PRINT B$
130 PRINT C$,D$, E$
140 PRINT F$
150 GOTO 1000
200 HOME
300 PRINT CHR$(4); "CLOSE TELEPHONE"
400 PRINT "THE NAME IS NOT ON FILE"
1000 END
```

TEST YOUR UNDERSTANDING 3

Use the above program to locate Mary Bell's number in the telephone file created in TEST YOUR UNDERSTANDING 1 and 2.

Example 3. Write a program which adds entries to the file **TELEPHONE.** The additions should be typed via **INPUT** statements. The program may assume that the file is on the diskette in drive 1.

Solution. Here's the program.

```
5 L=0
10 DIM A$(100),B$(100),C$(100),D$(100),E$(100),F$(100)
20 PRINT CHR$(4);"OPEN TELEPHONE"
25 PRINT CHR$(4);"READ TELEPHONE"
30 L=L+1
35 INPUT A$(L),B$(L),C$(L),D$(L),E$(L),F$(L)
40 IF A$(L) < > "END" THEN 30
45 L=L-1
50 PRINT CHR$(4);"ERA TELEPHONE"
51 PRINT CHR$(4);"OPEN TELEPHONE"
55 PRINT CHR$(4);"WRITE TELEPHONE"
60 FOR N=1 TO L
65 PRINT A$(N): PRINT B$(N): PRINT C$(N): PRINT D$(N):
   PRINT E$(N): PRINT F$(N)
70 NEXT N
75 PRINT CHR$(4);"CLOSE TELEPHONE"
100 HOME
110 PRINT "TYPE ENTRY:NAME,STREET ADDRESS,CITY, STATE,"
120 PRINT "ZIP CODE, TELEPHONE NO."
130 INPUT A$,B$,C$,D$,E$,F$
140 PRINT CHR$(4); "APPEND TELEPHONE"
150 PRINT CHR$(4); "WRITE TELEPHONE"
200 PRINT A$,B$, C$, D$, E$, F$
210 PRINT CHR$(4); "CLOSE TELEPHONE"
250 INPUT "ANOTHER ENTRY (Y/N)": Z$
260 IF Z$="Y" THEN 300
270 GOTO 400
300 HOME
310 GOTO 110
400 PRINT CHR$(4); "APPEND TELEPHONE"
410 PRINT CHR$(4); "WRITE TELEPHONE"
420 PRINT "END"
430 PRINT CHR$(4); "CLOSE TELEPHONE"
500 END
```

Writing "END" as the last data item indicates the end of the data file. This allows you to read to the end of the file and no further, thereby avoiding an error. Another way of handling the end of the file problem is to read the data items until an error actually does occur. To prepare for the expected error, place an **ONERR GOTO** statement before the point at which the error will occur. The statement should send the computer to a line containing a **RESUME** statement, which in turn sends the computer back to the next line after the one the error occurred at.

The next example presents a program useful for parents who wish to teach organizational skills to their children. Most children love to play with the computer. Here's a program which acts as an assignment book and monitors progress on homework. This program was designed for Jonathan Goldstein, a 10-year-old computer enthusiast.

Example 4. Write a program which sets up a data file for homework assignments. The child should enter the assignments, by subject, when returning from school. As assignments are completed, the child may check them off. The program should tell the child whether his homework is complete.

Solution. Your program will first ask if the assignment has been recorded before. If so, it'll be in a file. If not, the program will prompt the child to type in the assignments by subject, with prompts like

```
What is your math assignment?
```

The only rule is that assignments cannot have commas in their statements. The child types in the assignment followed by RETURN and the computer responds with the next subject. If there is no assignment, type RETURN. (You may customize the program by entering your own subjects.) After all assignments are entered, the computer asks the child if he wishes the assignments displayed. If the answer is yes, the computer then produces a list of all subjects with their corresponding assignments:

```
SUBJECT          ASSIGNMENT

MATH             P45 1-20
READING          CHAP 3
SPELLING         P80 SENTENCES
```

The computer now gives the child a chance to check off a completed assignment. The computer doesn't allow for forgetfulness. It asks for the subject, then displays the assignment and asks if, in fact, that assignment has been completed. If so, the next time the list of assignments is displayed, an X will appear beside completed assignments. Finally, the computer scans the list of assignments and decides whether all are complete. If so, it prints HOMEWORK DONE; if not, it prints HOMEWORK NOT DONE. Here's the program.

```
20 DIM B$(20),C$(20),D$(20)
30 HOME
40 PRINT "HAVE YOU ENTERED THIS ASSIGNMENT BEFORE?"
50 INPUT A$
60 IF A$="Y" THEN 70
65 GOTO 140
70 PRINT "SUBJECT";TAB(20) "ASSIGNMENT"
80 PRINT CHR$(4); "OPEN SCHED"
85 PRINT CHR$(4); "READ SCHED"
90 FOR J=1 TO 7
100 INPUT B$(J),C$(J),D$(J)
```

```
110 PRINT B$(J);TAB(15) C$(J);TAB(25) D$(J)
120 NEXT J
130 PRINT CHR$(4); "CLOSE SCHED"
135 GOTO 370
140 HOME
150 DATA "MATH", "SPELLING", "LANGUAGE","SOCIAL STUDIES"
160 DATA "READING", "SCIENCE", "CURRENT EVENTS"
170 FOR J=1 TO 7
180 READ B$(J)
190 PRINT "DO YOU HAVE ANY ";B$(J);" HOMEWORK TONIGHT?"
200 INPUT A$
210 IF A$="Y" THEN 220
215 GOTO 250
220 PRINT "WHAT IS THE ";B$(J);" ASSIGNMENT?"
230 INPUT C$(J)
250 NEXT J
260 PRINT "DO YOU WISH TO SEE YOUR ASSIGNMENTS?"
270 INPUT A$
280 IF A$="Y" THEN 300
290 GOTO 370
300 HOME
310 PRINT "SUBJECT";TAB(15) "ASSIGNMENT"
320 PRINT
330 FOR J=1 TO 7
340 PRINT B$(J); TAB(15) C$(J)
350 NEXT J
370 PRINT "DO YOU WANT TO CHECK OFF AN ASSIGNMENT?"
380 INPUT A$
400 IF A$="Y" THEN 410
405 GOTO 520
410 INPUT "SUBJECT ";B$
420 FOR J=1 TO 7
430 IF B$ <> B$(J) THEN 476
440 PRINT "IS THIS ASSIGNMENT DONE?"
450 PRINT C$(J)
460 INPUT A$
470 IF A$="Y" THEN D$(J)="X"
476 NEXT J
480 HOME
490 FOR J=1 TO 7
500 PRINT B$(J);TAB(15) C$(J);TAB(25) D$(J)
510 NEXT J
520 FOR J=1 TO 7
530 IF D$(J) <> "X" AND C$(J) <> "" THEN E$="W"
540 NEXT J
550 IF E$="W" THEN PRINT "HOMEWORK IS NOT DONE"
```

```
555 IF E$ <> "W" THEN PRINT "HOMEWORK DONE"
560 INPUT "DO YOU WISH TO CHECK OFF ANOTHER ASSIGNMENT ";A$
570 IF A$="Y" THEN GOTO 380
575 GOTO 577
577 PRINT CHR$(4); "DELETE SCHED"
580 PRINT CHR$(4); "OPEN SCHED"
581 PRINT CHR$(4); "WRITE SCHED"
590 FOR J=1 TO 7
600 PRINT B$(J), C$(J), D$(J)
610 NEXT J
620 PRINT CHR$(4); "CLOSE SCHED"
700 END
```

The files discussed so far are called **sequential files.** These files must be read in the exact order in which they are written. **DOS** also allows you to have **random access files.** These files let you read a given piece of data without reading all the data written ahead of it. For fast access to your data, random access files have a distinct advantage over sequential files. You can go anywhere you want in the file without starting at the beginning. You can jump from one place to another. Unfortunately, these files don't use space on the diskette very efficiently.

Random access files store data in records, each of a fixed length that you specify when you open the file. In order to do this, you've got to consider the application of the program to determine just how many characters you'll need for each entry. If you figure that none of your entries will contain more than 20 characters and you've allotted 25 just to be safe, everything should be fine. But your data will spill from one record into the next if one of your entries contains 30 characters. This means that you'll probably lose data.

Still, inefficient use of diskette space and risk aside, random access files are valuable. All you have to do is add a length parameter and a record number in the appropriate places. In the **OPEN** statement, include an L along with the number of characters you wish to allocate for each record. If the file name were **ADDRESS** your open statement might be:

```
PRINT CHR$(4);"OPEN ADDRESS,L 50"
```

The L sets the record length at 50 characters.

The record number enters the picture when you use the **DOS WRITE** statement to enter data. To store data in record number 15, for example, you'd use the statement

```
PRINT CHR(4)$;"WRITE ADDRESS, R 15"
```

where R stands for record. Assign the number 0 to the first record in a data file and continue numbering from that point.

To read the data in the hypothetical file, you use the READ command in the form

```
PRINT CHR(4)$; "READ ADDRESS, R 15"
```

After you hit RETURN, you should see all the data you've stored in record number 15. To go from R 15 to R5, for example, use the same statement substituting 5 for 15.

Exercises (answers on page 210)

1. Write a program creating a diskette data file containing the numbers 5.7, -11.4, 123, 485, and 49.
2. Write a program which reads the data file created in Exercise 1 and displays the data items on the screen.
3. Write a program which adds to the data file of Exercise 1 the data items 5,78, 4.79, and -127.
4. Write a program which reads the expanded file of Exercise 3 and displays all the data items on the screen.
5. Write a program which records the contents of checkbook stubs in a data file. The data items of the file should be as follows:

   check #, date, payee, amount, explanation

   Use this program to create a data file corresponding to your previous month's checks.
6. Write a program which reads the data file of Exercise 5 and totals the amounts of all the checks listed in the file.
7. Write a program which keeps track of inventory in a retail store. The inventory should be described by a data file whose entries contain the following information:

   item, current price, units in stock

   The program should allow for three different operations: Display the data file entry corresponding to a given item, record receipt of a shipment of a given item, and record the sale of a certain number of units of a given item.
8. Write a program which creates a recipe file to contain your favorite recipes.
9. (For teachers.) Write a program which maintains a student file containing your class roll, attendance, and grades.
10. Write a program maintaining a file of your credit card numbers and the party to notify in case of loss or theft.

6

An Introduction To Computer Graphics

In many applications, it's helpful to present data in pictorial form. By displaying numerical information in graphical form, it's often possible to develop insights and to draw conclusions which aren't immediately evident from the original data. In this chapter, you'll find procedures for using your Franklin computer to create various kinds of pictorial displays on the screen. Such procedures belong to the field of **computer graphics.**

6.1 Low Resolution Graphics Principles

There are three display modes: **text mode, low resolution graphics mode,** and **high resolution graphics mode.** The text mode is the one you have been using up to this point to display characters on the screen. Provided a Color Card (also known as a Video Phase Modulator) is installed in your Franklin or provided that you have a Franklin color computer, low resolution graphics mode lets you draw figures in as many as 16 colors. High resolution allows you to draw very detailed figures, but only in eight colors. Remember that you need a color monitor or a color television set as well. If you use a black and white monitor, you may use the graphics modes exactly as described below, but you will generate only black and white pictures.

You may select among these modes by using the following Floating Point BASIC commands.

TEXT = text mode
GR = low resolution graphics mode
HGR = high resolution graphics mode

When BASIC is started, the display is automatically in text mode. The above commands may be used to switch from one display mode to another, either within a program or via a keyboard command.

Text Mode

Unless you've installed an 80 column card, the video display of the Franklin computer produces 24 rows of 40 characters each. This gives you up to 24 x 40 or 960 possible character positions. These various character positions divide the screen into small rectangles, one rectangle corresponding to each character position.

Low Resolution Graphics Mode

Low resolution graphics mode divides the screen into the same number of columns as for text mode, but each text row is divided into two low resolution graphics rows. Figure 6-1 indicates the subdivision of the screen corresponding to the text mode.

The rows are numbered from 0 to 47, with row 0 being at the top of the screen and row 47 at the bottom. The columns are numbered from 0 to 39, with column 1 being at the extreme left and column 39 at the extreme right. Each rectangle on the screen is identified by a pair of numbers, indicating the row and column. For example, Figure 6-2 (page 116) shows the rectangle in the twelfth row and sixteenth column.

A graphics rectangle is always specified by giving the column number first.

The low resolution graphics command **GR** blacks out the top 20 lines of the screen, giving a 40x40 grid of rectangles for plotting. The bottom 4 lines may be used for text. The plotting color is automatically set to black by the **GR** command. This means that you won't be able to see anything you plot since it will appear as black on black. The first chore on entering low resolution graphics mode is to choose the color. Sixteen colors are possible, numbered as follows:

0–black	6–medium blue	12–green
1–magenta	7–light blue	13–yellow
2–dark blue	8–brown	14–aqua
3–purple	9–orange	15–white
4–dark green	10–grey	
5–grey	11–pink	

To set the color, use an instruction of the form

```
COLOR=ll
```

Once the color has been set, all subsequent plotting will be in that color until you give another **COLOR** instruction.

To light up the rectangle at position (x,y) (in the current color), use the graphics instruction

```
10 PLOT x,y
```

Here X denotes the column number and Y denotes the row number. To turn off the rectangle, plot it in color 0 (black).

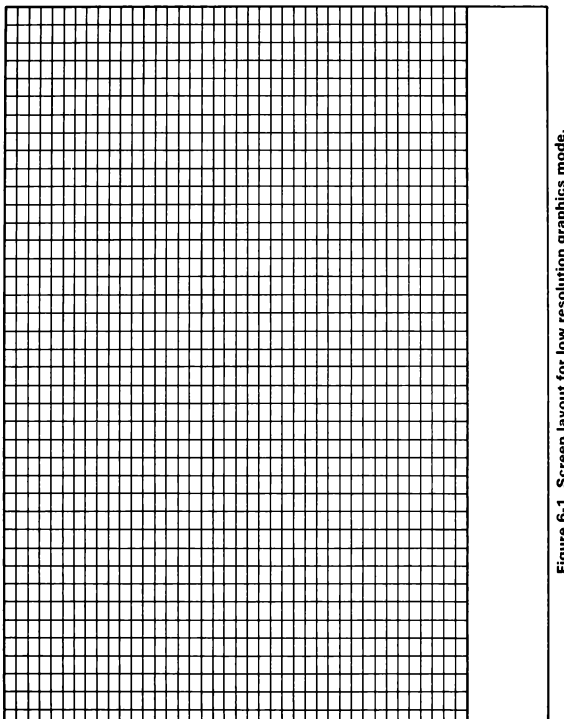


Figure 6-1. Screen layout for low resolution graphics mode.

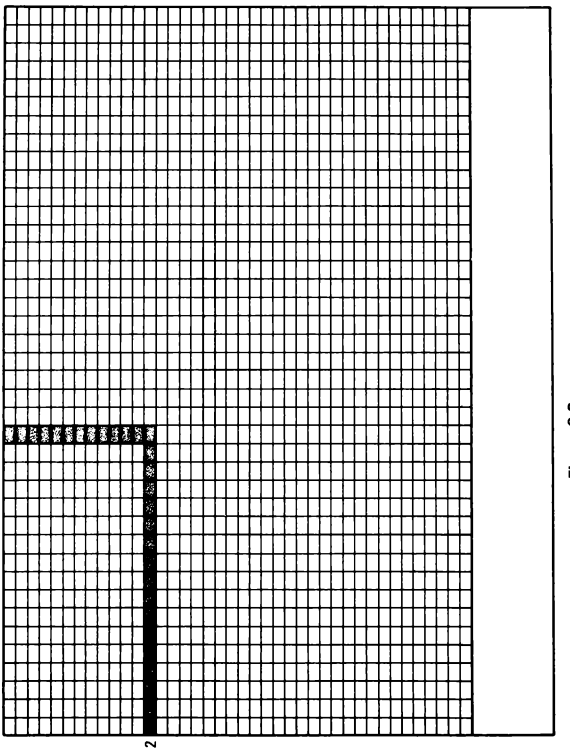


Figure 6-2.

TEST YOUR UNDERSTANDING 1 (answer on page 119)

Write a set of instructions to illuminate the graphics rectangle at row 18, column 22 in color 12.

TEST YOUR UNDERSTANDING 2 (answer on page 119)

Write a set of instructions to blink the graphics rectangle at row 18, column 22 in color 12.

Low resolution graphics has special commands for drawing horizontal and vertical lines. To draw a horizontal line from column 2, row 11 to column 20, row 11, use an instruction of the form

```
20 HLIN 2,20 AT 11
```

Similarly, to draw a vertical line from column 5, row 1 to column 5, row 20, use an instruction of the form

```
30 VLIN 1,20 AT 5
```

Example 1. Write a program which draws a dark green horizontal line across row 10 of the screen.

Solution.

```
10 GR
20 COLOR=4
30 HLIN 0,39 AT 10
40 END
```

Example 2. Write a program which draws a dark green vertical line in column 25 from row 5 to row 15. The program should blink the line 50 times.

Solution. The blinking effect may be achieved by repeatedly clearing the screen using **GR.** Here's the program.

```
10 GR
20 FOR J=1 TO 50:REM J CONTROLS BLINKING
30 COLOR=4
40 VLIN 5,15 AT 25
50 FOR K=1 TO 50: NEXT K
60 COLOR=0:VLIN 5,15 AT 25
70 FOR K=1 TO 50: NEXT K
80 NEXT J
90 END
```

TEST YOUR UNDERSTANDING 3 (answer on page 119)

Write a program to draw a vertical line from row 2 to row 20 in column 8.

Example 3. Draw a pair of x and y axes as shown in Figure 6-3. Draw the axes in light blue.

Solution.

```
10 GR
20 COLOR=7
30 HLIN 0,39 AT 39
40 VLIN 0,39 AT 0
50 END
```

Your program may determine the color of a given rectangle on the screen using the **SCRN** instruction. For example, **SCRN(7,3)** is equal to the color number of rectangle 7,3.

Exercises (answers on page 211)

Draw the following straight lines in magenta.

1. A horizontal line completely across the screen in row 18.
2. A vertical line completely up and down the screen in column 17.
3. A pair of straight lines which divide the screen into four equal squares.
4. Horizontal and vertical lines which convert the screen into a tic tac toe board.
5. A vertical line of double thickness from rows 1 to 24 in column 30.
6. A diagonal line going through the character positions (1,1), (2,2), ..., (24,24).
7. A horizontal line with "tick marks" as follows:

8. A vertical line with "tick marks" as follows:

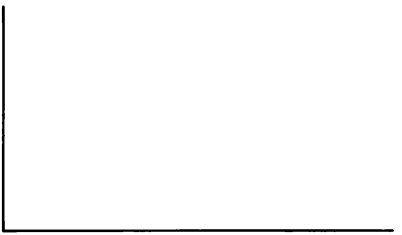


Figure 6-3.

ANSWERS TO TEST YOUR UNDERSTANDING 1,2, and 3

```
1:  10 GR
    20 COLOR=12
    30 PLOT 22,18
    40 END
2:  10 GR
    20 COLOR=12
    30 PLOT 22,18
    40 GOTO 10
    50 END
3:  10 GR
    20 COLOR=1
    30 VLIN 2,20 AT 8
    40 END
```

6.2 High Resolution Graphics

In principle, high resolution graphics on the Franklin computer works similarly to low resolution graphics, except that the screen is divided into 280 columns (0-279) and 160 rows (0-159). High resolution graphics mode is turned on with the instruction

```
HGR
```

Color in high resolution mode is set via the instruction

```
HCOLOR
```

However, only eight colors are available in high resolution graphics They are numbered 0-7. (Just which colors these are will depend on your particular television set.)

In high resolution graphics, point plotting is accomplished via the instruction

```
HPLOT
```

In addition to point plotting, high resolution graphics has provisions for defining and displaying elaborate shapes using shape tables stored in RAM. These shapes may be moved and even rotated using simple instructions. A complete discussion of high resolution graphics is beyond the scope of this book.

6.3 Computer Art

You can use your computer to create interesting pieces of graphics art, two forms of which deserve mention here.

The first arises when you give the computer free "artistic license" to create random patterns on the screen. Let's create a program which runs down the rows of graphics blocks and randomly illuminates some of them. You'll control the number of blocks to be illuminated by a "density factor" which you input. This factor will be a number between 0 and 1 and will be denoted by the variable D (for density). Your program will consider each graphics block separately, using the output of RND to make the decision on whether to illuminate the block or not. Namely, if RND is less than D, the block will be illuminated; if RND is at least D, then the block will not be illuminated. Here's your program.

```
 5 INPUT "DENSITY FACTOR"; D
 6 GR
10 FOR R=0 TO 39 : REM R=ROW NUMBER
20 FOR C=0 TO 39: REM C=COLUMN NUMBER
30 COLOR=INT(16*RND(1))
40 IF RND(1) < D THEN 60
50 PLOT C,R
100 NEXT C
110 NEXT R
200 END
```

You should run this program for assorted values of D. Some suggested values are D = .1, .5, and .7, respectively. Note that because of the unpredictability of the random number generator, the same value of D will usually yield quite different pictures on successive runs.

A second method of generating art is to use a graphics pad to effectively "trace" a picture. For example, suppose that your picture is a photograph.

Place a video display worksheet over the picture. Fill in all rectangles which touch the subject of the photograph. In this way, you'll create an impression of the subject which you may then display on the screen.

You might also be interested to know that some related equipment has recently become available to the computer hobbyist. In the brief description of tracing, you saw a rather laborious procedure. However, there are special devices called digitizing pads which enable you to trace a shape with an electronic "pen" and have the same shape transferred to the screen. In addition, there are the so-called light pens which enable you to touch a point on the screen and have the computer read the location of the point. This sort of device can be useful in creating computer art as well as in playing computer games.

Exercises (answers on page 212)

1. Run the above program three times for the value D = .4.
2. Run the above program for the values D = .1, .2, .3,. . ., 1.0. Can you predict the display for D = 1.0 in advance?
3. Create a computer impression of a member of your family using a large photograph. (5"x7" or larger will work best. Use high resolution graphics.)

7
Word Processing

7.1 What is Word Processing?

Nowhere does the revolutionary impact of microcomputers promise to be greater than in the area of word processing. In brief, a **word processor** is a device made by combining the traditional typewriter with the capabilities of the computer for storing, editing, retrieving, displaying, and printing information. It is no exaggeration to say that the traditional typewriter is now as obsolete as a Model T. Over the next decade or so, the typewriter will be completely replaced by increasingly sophisticated word processors.

The basic concept of a word processor is to use the microcomputer as a typewriter. However, instead of using paper to record the words, you use the computer memory. First, the words are stored in RAM. When you wish to make a permanent record of them, you store them on diskette as a data file. As you type, the text appears on the video display. This part of word processing is not revolutionary. The true power of a word processor doesn't come into play until you need to edit the data in a document. Using the power of the computer, you can perform the following tasks quickly and with little effort: Move to any point in the document; add words, phrases, sentences, or even paragraphs; delete portions of the text; move a block of text from one part of the document to another; insert "boiler-plate" information (standard pieces of text such as resumes or company descriptions) from another data file (for example, you could add a name and address from a mailing list); selectively change all occurrences of one word (say, "John") to another (say, "Jim"); or print the contents of a file according to a requested format.

All of the above operations are possible since the computer is able to manipulate strings in addition to numbers. Actually, your Franklin computer is equipped with a wide variety of commands to manipulate string data. In fact, you may turn your Franklin computer into quite a respectable word processor.

In this chapter, you'll see how to use the Floating Point BASIC instructions for string manipulation as well as for formatting output on a printer. Next, you'll find a discussion of the features available in word processing packages which you can purchase for your Franklin computer. Finally, to give you a taste of actual word processing, you'll build a rudimentary word processor which you can use to prepare letters, term papers, memos, or other documents.

7.2 Manipulating Strings

ASCII Character Codes

Each keyboard character is assigned a number between 0 and 255. The code number thus assigned is called the ASCII code of the character. For example, the letter "A" corresponds to the number 65, while the number 97 corresponds to the letter "a." Also included in this correspondence are the punctuation marks and other keyboard characters. As examples, 40 corresponds to the symbol " < " and 62 corresponds to the symbol " > ." Even the various control keys have corresponding numbers. For example, the space bar corresponds to the number 32, and the RETURN key to the number 13. Table 7-1 lists all of the printable characters and their corresponding ASCII codes. You'll find out about the various control codes in Section 7.4.

Note that most Franklin keyboards are capable of generating the symbols corresponding to all the ASCII codes. You may specify any ASCII symbols in your program, using the appropriate code numbers. For example, you can insert lower case letters in a diskette file or send lower case letters to the printer.

The computer uses ASCII codes to refer to letters and control operations. Any file, whether it's a program or data file, may be reduced to a sequence of ASCII codes. Consider the following address.

John Jones
2 S. Broadway

As a sequence of ASCII codes, it would be stored:

74,111,104,110,32,74,111,110,101,115,13,10
50,32,83,46,32,66,114,111,97,100,119,97,121,13

Note that the spaces are included (numbers 32) as are the carriage returns RETURN at the end of each line (number 13). ASCII codes let you describe any piece of text generated by the keyboard. This includes all formatting instructions like spaces, carriage returns, upper and lower case letters, and so forth. Moreover, once a piece of text has been reduced to a sequence of ASCII codes, it may also be faithfully reproduced on the screen or on a printer. This is the fundamental principle underlying the design of word processors.

Table 7-1. Franklin ASCII character codes.

Decimal Number	True ASCII	ASCII Graphic/Text	Screen POKE	Decimal Number	True ASCII	ASCII Graphic/Text	Screen POKE
0	NUL	(null)		64	@	@	0
1	SOH			65	A	A a	1
2	STX			66	B	B b	2
3	ETX	stop		67	C	C c	3
4	EOT			68	D	D d	4
5	ENQ			69	E	E e	5
6	ACK			70	F	F f	6
7	BEL	bell		71	G	G g	7
8	BS	(backspace)		72	H	H h	8
9	HT	tab		73	I	I i	9
10	LF	linefeed		74	J	J j	10
11	VT	(vert. tab)		75	K	K k	11
12	FF	(form-feed)		76	L	L l	12
13	CR	return		77	M	M m	13
14	SO	set text		78	N	N n	14
15	SI	set top		79	O	O o	15
16	DLE			80	P	P p	16
17	DC1	crsr-down		81	Q	Q q	17
18	DC2	rvs		82	R	R r	18
19	DC3	home		83	S	S s	19
20	DC4	delete		84	T	T t	20
21	NAK	delete line		85	U	U u	21
22	SYN	erase end		86	V	V v	22
23	ETB			87	W	W w	23
24	CAN	(cancel)		88	X	X x	24
25	EM	scroll up		89	Y	Y y	25
26	SUB			90	Z	Z z	26
27	ESC	escape		91	[	[	27
28	FS			92	\	\	28
29	GS	crsr-right		93	]	]	29
30	RS	(record-sep.)		94	↑	↑	30
31	US	(unit-sep.)		95	←	←	31
32	SP	space	32	96		space	32
33	!	!	33	97	a	!	33
34	"	"	34	98	b	"	34
35	#	#	35	99	c	#	35
36	S	S	36	100	d	$	36
37	%	%	37	101	e	%	37
38	&	&	38	102	f	&	38
39	'	'	39	103	g	'	39
40	(	(	40	104	h	(	40
41	)	)	41	105	i	)	41
42	*	*	42	106	j	*	42
43	+	+	43	107	k	+	43
44	,	,	44	108	l	,	44
45	-	-	45	109	m	—	45
46	.	.	46	110	n	.	46
47	/	/	47	111	o	/	47
48	0	0	48	112	p	0	48
49	1	1	49	113	q	1	49
50	2	2	50	114	r	2	50
51	3	3	51	115	s	3	51
52	4	4	52	116	t	4	52
53	5	5	53	117	u	5	53
54	6	6	54	118	v	6	54
55	7	7	55	119	w	7	55
56	8	8	56	120	x	8	56
57	9	9	57	121	y	9	57
58	:	:	58	122	z	:	58
59	;	;	59	123	;		59
60	<	<	60	124		<	60
61	=	=	61	125		=	61
62	>	>	62	126		>	62
63	?	?	63	127	DEL	?	63

TEST YOUR UNDERSTANDING 1 (answer on page 131)

Write a sequence of ASCII codes which will reproduce this ad:

FOR SALE: Beagle puppies. Pedigreed.
8 weeks. $125.

You may refer to characters by their ASCII codes by using CHR$. For example, CHR$(74) is the character corresponding to ASCII code 74 (upper case J); CHR$(32) is the character corresponding to ASCII code 32 (space). The **PRINT** instruction may be used in connection with CHR$. For example, the instruction

```
10 PRINT CHR$(74)
```

will display an upper case J in the first position of the first print field.

TEST YOUR UNDERSTANDING 2 (answer on page 131)

Write a program which will print the ad of TEST YOUR UNDERSTANDING 1 from its ASCII codes.

To obtain the ASCII code of a character, use the instruction **ASC**. For example, the instruction

```
20 PRINT ASC("B")
```

will print the ASCII code of the character "B", namely 66. In place of "B", you may use any string. The computer will return the ASCII code of the first character of the string. For example, the instruction

```
30 PRINT ASC(A$)
```

will print the ASCII code of the first character of the string A$. If A$ is the empty string "" (the string consisting of no characters), then using ASC(A$) will result in an ILLEGAL QUANTITY ERROR.

TEST YOUR UNDERSTANDING 3 (answer on page 131)

Determine the ASCII codes of the characters $, g, X, + without looking at the table.

You may compute the length of a string by using the **LEN** instruction. For example, **LEN("BOUGHT")** is equal to six since the string "BOUGHT" has six letters. Similarly, if A$ is equal to the string "Family Income", then

LEN(A$) is equal to 13. (The space between the words counts!) Here's an application of the **LEN** instruction.

Example 1. Write a program which inputs the string A$ and then centers it on a line of the display.

Solution. A line is 40 characters long, with the spaces numbered from 1 to 40. The string A$ takes up LEN(A$) of these spaces, so there are 40-LEN(A$) spaces to be distributed on either side of A$. The line should begin with half of the 40-LEN(A$) spaces, or with (40-LEN(A$))/2 spaces. Tab to column (40-LEN(A$))/2+1. Here's your program.

```
10 INPUT A$
20 HOME
30 PRINT TAB((40-LEN(A$))/2+1) A$
40 END
```

TEST YOUR UNDERSTANDING 4 (answer on page 131)

Use the program of Example 1 to center the string "THE FRANKLIN COMPUTER".

More About Strings

A string may consist of as many as 255 characters, but you cannot type more than 40 characters on a line. However, you may type strings containing more than 40 characters by continuing to type **without hitting the** RETURN **key**. When a line is filled, the computer will automatically place the next letter at the beginning of the next line. However, if you do not hit the RETURN key at the end of the line, then the next line will be a continuation of the first. It's necessary to have long strings if you wish to be able to print hard copy. Most line printers have at least 80 characters per line and some accommodate 232 character lines.

There are a number of operations which you can perform on strings. First, strings may be "added" (or, in computer jargon, "concatenated"). Suppose that you have strings A$ and B$, with A$ = "WORD" and B$ = "PROCESSOR". Then the sum of A$ and B$, denoted A$ + B$, is the string obtained by adjoining A$ and B$, namely:

"WORDPROCESSOR"

Note that no space is left between the two strings. To include a space, suppose that C$ = " ". C$ is the string which consists of a single space. Then A$ + C$ + B$ is the string

"WORD PROCESSOR"

TEST YOUR UNDERSTANDING 5 (answer on page 131)

If A$ = "4"and B$ = "7", what is A$ + B$?

The computer handles relations among strings in much the same way that it handles relations among numbers. For example, you say that the strings A$ and B$ are equal, written A$ = B$, provided that they consist of exactly the same characters, in the same order. Otherwise the strings are unequal, written A$ < > B$ or A$ > < B$. The notation A$ < B$ means that A$ precedes B$ in alphabetical order. (This is fine for strings consisting only of letters. For numbers or other characters, the ASCII codes of the characters are used to determine order.) Similarly, A$ > B$ means that A$ succeeds B$ in alphabetical order. For example, the following are true relations among strings:

"bear" < "goat"

"girl" > "boy"

The notation A$ > = B$ means that either A$ > B$ or A$ = B$. Simply, this means that A$ either succeeds B$ in alphabetical order or A$ and B$ are the same. The notation A$ < = B$ has a similar meaning.

Example 2. Write a program which alphabetizes the following list of words: egg, celery, ball, bag, glove, coat, pants, suit, clover, weed, grass, cow, and chicken.

Solution. Set up a string array A$(J) which contains these 13 words with B$ equal to the first word. Successively compare B$ with each of the words in the array. If any compared word precedes B$, you replace B$ with that word. At the end of the comparisons, B$ contains the first word in alphabetical order. Make this the first item in the array C$(J). Now repeat the process with the first word eliminated. This gives you the second word in alphabetical order, and so forth. Here's your program.

```
10 DIM A$(20),D$(20),C$(20)
20 DATA EGG,CELERY,BALL,BAG,GLOVE,COAT
30 DATA PANTS,SUIT,CLOVER,WEED,GRASS
40 DATA COW,CHICKEN
50 FOR J=1 TO 13
60 READ A$(J):REM SET UP ARRAY A$
70 NEXT J
80 FOR K=1 TO 13:REM FIND KTH WORD IN ORDER
85 REM 90-200 CREATE AN ARRAY D$ CONSISTING OF
87 REM THE WORDS YET TO BE ALPHABETIZED
90 L=1
100 FOR J=1 TO 13
110 E=0
120 FOR M=1 TO 13
```

```
130 IF A$(J)=C$(M) THEN E=1
140 NEXT M
150 IF E=0 THEN 160
155 GOTO 200
160 D$(L)=A$(J)
170 L=L+1
200 NEXT J
210 B$=D$(1)
220 FOR L=1 TO 13-K+1
230 IF D$(L) < B$ THEN 250
235 GOTO 300
250 B$=D$(L)
300 NEXT L
400 C$(K)=B$
410 NEXT K
500 FOR K=1 TO 13
510 PRINT C$(K)
520 NEXT K
600 END
```

This program can be modified to make a program alphabetizing any collection of strings. You'll find the details in the exercises.

It's possible to dissect strings using the three instructions **LEFT$**, **RIGHT$**, and **MID$**. These instructions allow you to construct a string consisting of a specified number of characters taken from the left, right, or middle of a designated string. Consider the instruction:

```
10 LET A$=LEFT$("LOVE",2)
```

The string A$ consists of the two leftmost characters of the string "LOVE". That is, A$ = "LO". Similarly, the instructions

```
20 LET B$="TENNIS"
30 LET C$=RIGHT$(B$,3)
```

set C$ equal to the string consisting of the three rightmost letters of the string B$, namely C$ = "NIS". Similarly, if A$ = "REPUBLICAN", then the instruction

```
40 LET D$=MID$(A$,5,3)
```

sets D$ equal to the string which consists of three characters starting with the fifth character of A$, which is D$ = "BLI". These last three instructions are absolutely fundamental in the design of word processors.

TEST YOUR UNDERSTANDING 6 (answer on page 131)

Determine the string constant

RIGHT$(LEFT$("COMPUTER",4),3)

In manipulating strings, it's important to recognize the difference between numerical data and string data. The number 14 is denoted by 14; the string consisting of the two characters 14 is denoted "14". The first is a numerical constant and the second a string constant. You can perform arithmetic using the numerical constants. However, you cannot perform any of the character manipulation supplied by the instructions **RIGHT$**, **MID$**, and **LEFT$**. Such manipulation may only be performed on strings. How can you perform character manipulation on numerical constants? BASIC provides a simple method. First convert the numerical constants to string constants by using **STR$**. For example, the number 14 may be converted into the string "14" using the instruction:

```
10 LET A$=STR$(14)
```

As a result of this instruction, A$ has the value "14". As another example, suppose that the variable B has the value 1.457. **STR$(B)** is then equal to the string "1.457".

To convert strings consisting of numbers into numerical constants, use **VAL**. Consider this instruction:

```
20 LET B=VAL("3.78")
```

This instruction sets B equal to 3.78.

TEST YOUR UNDERSTANDING 7 (answer on page 131)

Suppose that A$ equals "5 PERCENT" and B$ equals "758.45 DOLLARS". Write a program which starts from A$ and B$ and computes five percent of $758.45.

Exercises (answers on page 212)

1. Use the program of Example 2 (page 128) to alphabetize the following sequence of words: justify, center, proof, character, capitalize, search, replace, indent, store, and password.
2. Write a program to form the following string into lines consisting of 80 characters each.

Word processing will revolutionize the office. Already, millions of word processing systems are in use. By the end of the decade, the typewriter will be totally obsolete. Word processing systems will increase in sophistication.

3. Write a program which rewrites the addition problem 15 + 48 + 97 = 160 in the form

$$\begin{array}{r} 15 \\ 48 \\ 97 \\ \hline 160 \end{array}$$

4. Write a program which inputs the string constants "$6718.49" and "$4801.96" and calculates the sum of the given dollar amounts.

ANSWERS TO TEST YOUR UNDERSTANDING 1,2,3,4,5,6, and 7

1: 70,79,82,32,83,65,76,69,58,32,66,101,97,103,108,101,32, 112,117,112,112,105,101,115,46,32,13,10,56,32,119,101, 101,107,115,46,32,36,49,50,53,46,13,10

2:
```
5 DIM A(43)
10 DATA 70,79,.....(insert data from 1)
11 DATA........
12 DATA........
20 FOR J=1 TO 43
30 READ A(J)
40 PRINT CHR$(A(J));
50 NEXT J
60 END
```

3:
```
10 DATA $,g,X,+
20 FOR J=1 TO 4
30 READ A$(J)
40 B(J)=ASC(A$(J))
50 PRINT A$(J), B(J)
60 NEXT J
70 END
```

4: Type RUN followed by the given string.

5: 47

6: "omp"

```
7:  10 A$="5 PERCENT":B$="758.45 DOLLARS"
    20 A=VAL(A$):B=VAL(B$)
    30 PRINT A$;"OF";B$;" IS"
    40 PRINT A*B*.01
    50 END
```

7.3 Printer Controls and Form Letters

Let's review what's been covered up to this point concerning the printer.

The printer is controlled by a circuit board placed in one of the slots of the Franklin computer. For clarity, let's assume that the card controlling the printer is placed in slot 1. The instruction:

```
10 PR#1
```

directs all further **PRINT** statements to the printer in addition to the screen. To go back to displaying the results of **PRINT** statements only on the screen, you may use the instruction:

```
20 PR#0
```

The printer accepts a stream of ASCII character codes. Some of these correspond to letters and symbols to be printed, and some correspond to control characters which make the printer perform various non-print functions (such as carriage return, line feed, space to the top of the next page, and so forth). To the printer, a line consists of a sequence of printable characters followed by RETURN, which actually tells the printer two things. First, it causes a carriage return; second, it causes the paper to advance one line (a so-called **line feed**).

A typical printer has controls for setting a number of different parameters. There are usually switches to control line width, margins, the number of lines per inch, the number of lines per page, and the size of the characters. Your printer manual should show you the correct procedures for setting these switches. (In the case of some printers, these various settings are controlled by sequences of characters from the computer. These so-called **ESCAPE** sequences are printed as if they were actual data. However, the printer recognizes them as commands and does not print them.)

Form Letters

You may use the string manipulating capability of your computer to prepare form letters that look like genuine correspondence. Let's illustrate the technique by preparing the following form letter.

April 1, 1983

Dear :

All of us at Neighborhood Building Supplies, Inc. have appreciated your patronage in the past. We are writing to let you know that we will move our store to 110 S. Main St., effective July 1. You will find the new store larger and more convenient than the old. In addition, we will now stock a more extensive line of energy-efficient doors and windows. We look forward to your continued patronage. Please let us know if we may be of assistance in your building plans.

Cordially yours,

Samuel Gordon,
President
Neighborhood Building Supplies, Inc.

Suppose that this letter is to go out to 100 customers on a mailing list. Also assume that the mailing list is maintained in a data file on diskette and that each address is stored as four strings, corresponding to the name of the individual, company name, street address, and city-state-zip code. Finally, suppose that the name of the file is CUSTOMER. Write a program to produce the desired stack of 100 letters.

Your program will consist of two parts. The first will allow you to type in the body of the letter. The various lines of the letter will be typed exactly like you were typing on a typewriter. The computer will use a string array A$(J) to store the body of the letter, with A$(1) holding the first line, A$(2) the second, and so forth. You can indicate to the computer that the body of the text is complete by typing % followed by RETURN.

As soon as the character % is recognized, the program goes into its second phase, namely the actual generation of the letters. The program opens the address file for output. One by one it reads the address entries. After reading a given entry, it prints the date at the top of the letter, followed by the address. Next, the program determines the last name of the addressee from the first line of the address and inserts it after the "Dear." Finally, the body of the letter is printed. Here is the program which accomplishes all of this.

```
10 DIM A$(100)
20 PRINT "AFTER EACH ? TYPE ONE LINE OF THE LETTER,"
30 PRINT "FOLLOWED BY RETURN"
40 PRINT "TO END LETTER, TYPE % FOLLOWED BY RETURN"
50 J=1 : REM J IS THE NUMBER OF LINES
60 INPUT A$(J)
70 IF A$(J)="%" THEN 100
80 J=J+1: REM NEXT LINE
90 GOTO 60
100 PRINT CHR$(4);"OPEN CUSTOMER"
110 PRINT CHR$(4);"READ CUSTOMER"
115 FOR N=1 TO 100: REM N=CUSTOMER #
120 INPUT B$(1,N), B$(2,N), B$(3,N), B$(4,N)
130 NEXT N :REM GO TO NEXT CUSTOMER
140 PRINT CHR$(4);"CLOSE CUSTOMER"
150 PR#1: REM ADDRESS PRINTER
155 FOR N=1 TO 100
160 PRINT A$(1): REM PRINT DATE
170 PRINT: REM PRINT BLANK LINE
175 REM 150-180 PRINT ADDRESS
180 PRINT B$(1,N)
181 PRINT B$(2,N)
182 PRINT B$(3,N)
183 PRINT B$(4,N)
190 REM FIND LAST NAME OF ADDRESSEE
200 LET L=LEN(B$(1,N))
210 FOR K=0 TO L-1
220 LET C$=MID$(B$(1,N),1,L-K)
230 IF C$=" " THEN 300
235 GOTO 240: REM TEST FOR SPACE BETWEEN WORDS
240 NEXT K
300 LET D$=RIGHT$(B$(1,N),K): D$=REM LAST NAME
310 PRINT "Dear Mr. "; D$; ","
320 FOR M=2 TO J :REM PRINT BODY OF LETTER
330 PRINT A$(M)
340 NEXT M: REM GO TO NEXT LINE OF LETTER
350 NEXT N : REM GO TO NEXT CUSTOMER
360 PR#0: TURN SCREEN ON
400 END
```

Note that for the above program to work properly, you should type the date in as the first line of the letter. You should not type in the line which begins "Dear". The program generates this line for you. Also note that this program always addresses the customer as "Mr." This will insult a certain number of your customers. Suppose that your customer entries in the address list are labelled with "Mr.," " Mrs.," or "Ms." preceding the name. Can you

modify the above program to insert the correct title in the salutation of the letter?

The above program was used in the context of a specific letter. Please note that the program may be used to generate any set of form letters from an address list. In the exercises, you'll find some modifications which you can use to generate invoices or other correspondence with variable text in the body of the letter.

Exercises

1. Add to the form letter of the text a second page. At the top of the page should be a date and the page number. The title should be "OPENING SPECIAL." The text should consist of the following message:

```
This letter is being sent only to our most
valued customers! Bring this coupon with
you for a 10 percent discount on any order
placed in the month of JULY, 1983.
```

2. Change the form letter of the text so that in the third and fourth sentences, the name of the addressee is used. (For example, "Ms. Thomas, you will find ...")

3. Write a program that prints invoices. Assume that the invoices are stored in a file called **INVOICE** where a particular invoice contains for each item shipped: quantity, price, item description (limited to 15 characters), and total cost. Assume that the invoice entry starts with a four-string entry giving the customer name, address, and date. The file entry for a given invoice ends with the character %. You may assume that the first entry in the file is the number of invoices contained in the file. Your program should print out invoices corresponding to all entries in the file.

4. Suppose that the file **INVOICE** of Exercise 3 contains only a customer identification number rather than a customer name and address. Suppose that the whole list of customer addresses contains customer identification numbers. Modify the program of Exercise 3 so that it locates the customer name and address automatically.

5. Assume that a change in local ordinances now allows you to not charge local sales tax to any customer who lives outside the city limits. Suppose that the city consists of ZIP CODE 91723. Modify the program of the text so that it checks the ZIP CODE of the customer. For customers not in ZIP CODE 91723, insert the following paragraph in the letter:

```
Good news! You will no longer be charged
local sales tax, in accordance with the
change in local ordinances. This will
yield even further savings from our
already low prices.
```

7.4 Using Your Computer As a Word Processor

So far in this chapter, you've been introduced to the various text manipulation features of the Franklin computer. You've also found out a little about word processing which you can accomplish using homegrown programs. However, your computer is capable of quite a bit more than a beginning programmer can expect to accomplish at this stage of learning. However, you might be interested in a description of some of the word processing you can expect your computer to accomplish using commercially available software.

At its most basic level, you use a word processing system like you would use a typewriter. Suppose that you wish to prepare a document. You would turn on the computer and run the word processing program. The program first asks for the type of work you would like to perform. Possibilities include: type in a new document, edit an old document, save a document on diskette, or print a document. You'd select the first option. Next you'd describe various format parameters to the word processor: line width, number of characters per inch, number of lines per page, spacing between lines, and so forth.

You would then type the document exactly like you would on a typewriter. There are several huge exceptions, however!

First of all, don't worry about carriage returns. The word processor takes care of forming lines. It accepts the text you type, decides how much can go on a line, forms the line, and displays it. Any text left over is automatically saved for the next line. The only function of the carriage return is to indicate a place where you definitely want a new line, such as at the end of a paragraph.

A second advantage of a word processor is in correcting errors. To correct an error, move the cursor to the site of the mistake, give a command to erase the erroneous letter(s) or word(s), and type in the replacements. Of course, such action will generally destroy the structure of the lines. (Some lines may now be too long and others too short.) By using a simple command, it's possible to "reform" the lines according to the requested format.

Typically, a word processor has commands which enable you to scroll through the text of a document to look for a particular paragraph. Some

word processors even allow you to mark certain points so that you may turn to them without a visual search.

When the document is finally typed to your satisfaction, you give the computer an instruction which saves a copy of it on diskette. At a future time, you may recall the document, and add to it at any point (even within the bodies of paragraphs!). Typically, word processors have certain "block operations" which allow you to "mark" a block and then either delete it, copy it, or move it to another part of the document. You may also insert other documents into the current document. This is convenient, for example, in adding boiler plate, such as resumes, to your document. You may even use the block operations to alter boiler plate to fit the special needs of the current document.

Construct your document in as many sessions as you wish. When your diskette finally contains the document as you want it, you finally give the instruction to print. Your printer will now produce a finished, error-free copy of the document.

As if the above were not enough of an improvement over the conventional typewriter, the typical word processor can do even more. The features available depend, of course, on the word processor selected. Here are some of the goodies to look for:

Global Search and Replace. Suppose you wish to resubmit your document to another company, Acme Energetics. In your original document, you included numerous references to the original company, Jet Energetics. A global search and replace feature allows you to instruct the computer to replace every occurrence of a particular phrase with another phrase. For example, you could replace every occurrence of "Jet Energetics" with "Acme Energetics." Global search and replace can be even more sophisticated. In some systems, the word processor can be instructed to ask you whether or not to make each individual change. Another variation is to instruct the word processor to match any capitalization in the phrases replaced.

Centering. After typing a line you may center it using a simple command.

Boldface. You may print certain words in darker type.

Underscore. You may indicate emphasis by underscoring portions of text.

Subscripts and Superscripts. You may indicate printing of subscripts and superscripts. This is extremely useful for scientific typing.

Justification. You may instruct the word processor to "justify" the right hand margins of your text, so that the text always ends exactly at the end of a line. This is possible only if you have a printer which is capable of spacing in increments smaller than the width of a single letter.

Spelling Correction. There now exist a number of spelling correction programs which compare words of your document against a dictionary (sizes range from 20,000 to 70,000 words). If the program doesn't find a match, it asks you if the word is spelled correctly and gives you an opportunity to add the word to the dictionary. In this way the output of a word processor can be proofread by computer.

7.5 A Do-It-Yourself Word Processor

It's really quite impractical for you to build your own word processor. For one thing, such a program is long and complicated. Moreover, if you write in BASIC, the operation of the program will tend to be rather slow. An efficient word processor will almost always be written in machine language. Nevertheless, in order to acquaint you with a few of the virtues of word processing, let's build a word processor anyway!

This word processor will be line oriented. You'll type in each line just as if you're typing it on a typewriter. At the end of each line, you'll give a carriage return by typing [RETURN]. The Jth line will be stored in the string variable A$(J).

This word processor will have five modes. In the first mode, you input the text of your document. This operation will proceed exactly as if you were typing on a typewriter. At the beginning of each line, the word processor will display a ? You type your line after the question mark. Terminate the line with [RETURN] . In order to indicate that you don't wish to type any more lines, type % followed by [RETURN] .

A second mode allows you to save your document. For the purposes of this word processor, let's assume that you have a diskette file. The program then saves your document as a data file under a file name requested by the program. The first item in a document file will always be the number of lines in the document. This quantity will be denoted by the variable L. Next come the lines of the document: A$(1), A$(2), ... , A$(L).

A third mode lets you produce a draft version of the document. In this mode, the document is printed with each line preceded by its line number. The line numbers allow you to easily identify lines having errors. Note that in order to print a document, you must first save it on the diskette.

A fourth mode allows document editing. To correct errors, you identify the line by number and retype the line. To end the edit session type % followed by [RETURN]. This will bring you back to the beginning of the program, but you'll still be working on the same document. After ending an edit session, your customary next action should be to save the document. The fifth and final mode allows printing of a final draft of a document.

When the word processor is first run, you'll see the following prompt:

```
WORD PROCESSING PROGRAM
CHOOSE ONE OF THE FOLLOWING MODES

     INPUT TEXT(I)
     DISPLAY DRAFT (DD)
     PRINT FINAL DRAFT (PF)
     SAVE FILE (S)
     EDIT (E)
     QUIT (Q)
```

In response, you type one of I, DD, PF, S, E, or Q, followed by RETURN. If you choose I, the screen will be cleared and you may begin typing your document. For the other modes, there are prompts to tell you what to do. Here's a listing of the program.

```
5 DIM A$(150)
10 PRINT "WORD PROCESSING PROGRAM"
20 PRINT "CHOOSE ONE OF THE FOLLOWING MODES"
30 PRINT ,"INPUT TEXT(I)"
40 PRINT ,"DISPLAY DRAFT(DD)"
50 PRINT ,"PRINT FINAL DRAFT(PF)"
60 PRINT ,"SAVE FILE(S)"
70 PRINT ,"EDIT(E)"
80 PRINT ,"QUIT(Q)"
90 INPUT X$
100 IF X$="I" THEN 1000
110 IF X$="DD" THEN 2000
120 IF X$="PF" THEN 3000
130 IF X$="S" THEN 4000
140 IF X$="E" THEN 5000
150 IF X$="Q" THEN 6000
160 GOTO 90: REM IF X$ DOES NOT MATCH ANY OF THE PROMPTS
1000 L=1
1010 INPUT A$(L)
1020 IF A$(L)="%" THEN L=L-1:GOTO 10
1030 L=L+1
1035 IF L<=150 THEN 1010
1040 IF L>150 THEN PRINT "DOCUMENT TOO LARGE"
1050 GOTO 10
2000 INPUT"DOCUMENT NAME";Y$
2010 PRINT CHR$(4);"OPEN" Y$
2015 PRINT CHR$(4);"READ" Y$
2020 INPUT L
2030 FOR K=1 TO L
2040 INPUT A$(K)
2050 PRINT K;">";A$(K)
2060 NEXT K
```

```
2070 PRINT CHR$(4);"CLOSE"; Y$
2090 GOTO 10
3000 INPUT "DOCUMENT NAME";Y$
3010 PRINT CHR$(4);"OPEN"; Y$
3015 PRINT CHR$(4);"READ"; Y$
3020 INPUT L
3030 FOR K=1 TO L
3040 INPUT A$(K)
3050 NEXT K
3060 PRINT CHR$(4);"CLOSE"; Y$
3070 PR#1
3080 FOR K=1 TO L
3090 PRINT A$(K)
3100 NEXT K
3110 PR#0
3120 GOTO 10
4000 INPUT"DOCUMENT NAME";Y$
4010 PRINT CHR$(4);"OPEN";Y$
4015 PRINT CHR$(4);"WRITE";Y$
4020 PRINT L
4030 FOR K=1 TO L
4040 PRINT A$(K)
4050 NEXT K
4060 PRINT CHR$(4);"CLOSE"; Y$
4070 GOTO 10
5000 INPUT"DOCUMENT NAME"; Y$
5010 PRINT CHR$(4);"OPEN";Y$
5015 PRINT CHR$(4);"READ";Y$
5020 INPUT L
5030 FOR K=1 TO L
5040 INPUT A$(K)
5050 NEXT K
5055 PRINT CHR$(4); "CLOSE ";Y$
5060 INPUT "NUMBER OF LINE TO EDIT";Z
5070 HOME
5080 PRINT A$(Z)
5090 INPUT "TYPE CORRECTED LINE";A$(Z)
5100 IF A$(Z) <>"%" THEN 5060
5110 GOTO 10
6000 END
```

You should use this program to type a few letters. You will find it a big improvement over a conventional typewriter. Moreover, this will probably whet your appetite for the more advanced word processing features we described in the preceding section.

Exercises

1. Modify the word processor to allow input of line width. (You will not be able to display lines longer than 40 characters on a single line. However, string variables may contain up to 255 characters.)
2. Modify the word processor so that you may extend a line. This modification should let your corrected line spill over into the next line of text. The program should then correct all of the subsequent lines to reflect the addition.
3. Modify the word processor to allow deletions from lines. Subsequent lines should be modified to reflect the deletion.

8
Computer Games

In the last few years computer games have captured the imaginations of millions of people. In this chapter, you'll build several computer games which utilize both the random number generator and the graphics capabilities of the Franklin computer. In many games, you need a clock to time moves. The Franklin does not come with a built-in clock, but there are several available from outside vendors.

8.1 Telling Time With Your Computer

You may equip your Franklin computer with a clock (a **real-time clock** in computer jargon) which lets your programs take into account the time of day (in hours, minutes, and seconds) and the date (day of the week, date, and month). You can use this feature for many purposes, such as keeping a computer-generated personal calendar (see Example 1) or timing a segment of a program (see Example 2). There are several clocks available in the form of circuit boards. You insert the circuit board into one of the unused slots of your Franklin. In this chapter, for the sake of clarity, assume that such a printed circuit board has been inserted into slot 3.

Reading the Real-Time Clock

Many real-time clocks keep track of six pieces of information in the following order:

Month: MO (1-12)
Day of Week: DW (0-6), Sunday = 0
Date: DT (1-31)
Hours: HR (0-23)
Minutes: MN (0-59)
Seconds: SEC (0-59)

The date and time are displayed in the following format:

```
02 03 15 14 38 27
```

The above display corresponds to Wednesday, February 15, at 27 seconds after 2:38 PM. Note that the hours are counted using a 24-hour clock, with 0 hours corresponding to midnight. Hours 0-11 correspond to AM and hours 12-23 correspond to PM.

To obtain the reading of the clock in numerical format, use the following sequence of BASIC commands:

```
10 PR#3: IN#3
20 INPUT "#";MO,DW,DT,HR,MN,SEC
```

The variables MO, DW, DT, HR, MN, and SEC will then contain integers equal to the current reading of the various entries of the clock. For example, to display the current time on the screen, use the sequence of commands

```
10 PR#3: IN#3
20 INPUT "#";MO,DW,DT,HR,MN,SEC
30 PR#0: IN#0
40 PRINT MO,DW,DT,HR,MN,SEC
```

TEST YOUR UNDERSTANDING 1

Display the current time and date.

Setting the Clock

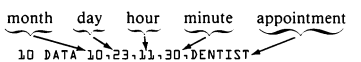
You have an opportunity to set the clock when you install it. A battery keeps the clock running when the computer is turned off. The manual that comes with the clock should show you how to set it.

TEST YOUR UNDERSTANDING 2

Write a program which continually displays the correct time on the screen.

Example 1. Use the real-time clock to build a computerized appointment calendar.

Solution. Enter appointments in **DATA** statements, with one **DATA** statement for each appointment. Let the **DATA** statements be arranged in the following format:



The appointments will be numbered by a variable J. For appointment J, store the five pieces of data in the variables A(J),B(J),C(J),D(J),E$(J), respectively. Let's allow for 300 appointments. Dimension each of the variables for an array of size 301.

1. Read the various appointments in the **DATA** statements. It will stop reading when there's no more data to read.
2. Ask for the current date and time data.
3. Determine today's appointments.

Here's the program:

```
10 DIM A(300),B(300),C(300),D(300),E$(300)
15 ONERR GOTO 100 : REM IF OUT OF DATA GO TO 100
20 REM 30-60 READ THE DATA STATEMENTS IN 1000-1300
30 J=1
40 READ A(J),B(J),C(J),D(J),E$(J)
50 J=J+1 : REM FINAL VALUE OF J=# APPOINTMENTS+1
60 GOTO 40
100 RESUME : REM END ERROR CONDITION AND GO TO 105
105 REM 110-160 ASK FOR CURRENT TIME AND DATE
110 PR#3:IN#3
115 INPUT "#";MO,DW,DT,HR,MN,SEC
120 PR#0:IN#0
200 PRINT "REQUEST SCHEDULE FOR MONTH,DAY"
210 PRINT "FOR TODAY, ENTER 0,0"
220 PRINT "FOR TOMORROW, ENTER -1, -1"
230 INPUT X,Y: REM X=MONTH, Y=DAY
240 IF X > 0 THEN 300
245 REM IF TODAY OR TOMORROW, FIND DATE AND DAY
250 X=MO-X: Y=DT-Y
300 FOR K=1 TO J-1
310 IF A(K)=X THEN 400: REM IS APPT. FOR RIGHT MONTH?
320 GOTO 510: REM IF NOT GO TO NEXT APPT.
400 IF B(J)=Y THEN 500: REM IS APPT. FOR RIGHT DAY?
410 GOTO 510: REM IF NOT GO TO NEXT APPT.
500 PRINT A(K),B(K),C(K),D(K),E$(K)
510 NEXT K
10000 DATA
10001 DATA
10002 DATA
.
.
.
```

```
10299 DATA
11000 END
```

To use the above program, type appointments into the data lines as you make them. You can store the current list of appointments along with the program. As new appointments are made, you can recall the program and add corresponding data lines. As appointments become obsolete, they may be removed from the list using the **DEL** command.

A few comments are in order concerning the **ONERR GOTO** and **RESUME** statements. At any given moment, you have no way of knowing how many appointments are in the calendar. Therefore, you have no way of knowing in advance how many of the data lines in 10000-10299 you must read. If you attempt to read past the last data line, you will get an **OUT OF DATA** error. The **ONERR GOTO** statement allows you to handle this error, when it occurs, without terminating program execution. In the above program, you have instructed the computer to respond to an error (namely the **OUT OF DATA** error you expect) by going to line 110. In this line, tell the computer to **RESUME**; that is, to continue execution of the program with the next line after 105, just as if the error never occurred. This procedure allows you to read data until there is no more.

The above program assumes that you store the appointments in **DATA** statements. Of course, it's perfectly possible to store the appointments in a data file on either cassette or diskette. Leave it as an exercise to modify the above program to allow for use of such files.

Example 2. In Example 6 of Chapter 2, Section 6, (page 46) you developed a program to test mastery in the addition of two-digit numbers. Redesign this program to allow 15 seconds to answer the question.

Solution. Let's use the real-time clock. After a particular problem has been given, the number of seconds will be recorded as

```
3600*HR+60*MN+SEC
```

and the program will perform a loop which continually compares this number with the current number of seconds. The loop will end when the two numbers differ by 15 (15 seconds has elapsed). Then the program will print out "TIME'S UP. WHAT IS YOUR ANSWER?" Here's the program. Lines 50 and 60 contain the loop.

```
10 FOR J=1 TO 10 : REM LOOP TO GIVE 10 PROBLEMS
20 INPUT "TYPE TWO 2-DIGIT NUMBERS"; A,B
30 PRINT "WHAT IS THEIR SUM?"
40 PR#3:IN#3
45 INPUT "#";MO,DW,DT,HR,MN,SEC
47 A=3600*HR+60*MN+SEC
49 INPUT"#";MO,DW,DT,HR,MN,SEC
50 B=3600*HR+60*MN+SEC
55 IF B-A=15 THEN GOTO 100: REM COUNT 15 SECONDS
```

```
60 GOTO 49
100 PR#0:IN#0
110 INPUT "TIME'S UP! WHAT IS YOUR ANSWER?";C
120 IF A+B=C THEN 200
130 PRINT "SORRY. THE CORRECT ANSWER IS",A+B
140 GOTO 500 : REM GO TO THE NEXT PROBLEM
200 PRINT "YOUR ANSWER IS CORRECT! CONGRATULATIONS"
210 LET R=R+1 : REM INCREASE SCORE BY 1
500 NEXT J: REM GO TO NEXT PROBLEM
600 PRINT "YOUR SCORE IS",R,"CORRECT OUT OF 10"
700 PRINT "TO TRY AGAIN, TYPE RUN"
800 END
```

TEST YOUR UNDERSTANDING 3

Modify the above program so that it allows you to take as much time as you like to solve a problem, but keeps track of elapsed time (in seconds) and prints out the number of seconds used.

Exercises

1. Set the clock with today's date and the current time.
2. Print out the current time on the screen.
3. Write a program which prints out the date and time at one-second intervals.
4. Write a program which prints out the date and time at one-minute intervals.
5. Set up the appointment program of Example 1 (page 144) and use it to enter a week's worth of appointments.
6. Modify the appointment program of Example 1 so that it prints out the appointments on a per hour basis.
7. Modify the addition tester program so that it allows a choice of four levels of speed: Easy (2 minutes), Moderate (30 seconds), Hard (15 seconds), and Whiz Kid (8 seconds).
8. Write a program which can be run for an entire day and at the start of each hour prints out the appointments for that hour. (Such a program would be useful in a doctor's or dentist's office.)

8.2 Blind Target Shoot

The object of this game is shoot down a target on the screen by moving your cursor to hit the target. The catch is that you only have a two-second look at your target! The program begins by asking if you're ready. If so, you type

READY. The computer then randomly chooses a spot to place the target. It lights up the spot for two seconds. The cursor is then moved to the upper left position of the screen (the so-called "home" position). You must then move the cursor to the target, based on your brief glimpse of it. You have 10 seconds to hit the target. (See Figure 8-1.)

Your score is based on your distance from the target, as measured in terms of the moves it would take to get to the target from your final position. Here's the list of possible scores:

Distance From Target	Score
0	100
1 or 2	90
3 to 5	70
6 to 10	50
11 to 15	30
16 to 20	10
over 20	0

To input cursor moves while the program is running, use the optional game paddles which may be connected to the Franklin. By rotating the paddles, you input a number to the computer. This number is between 1 and 255. Set up the following correspondence between paddle input and cursor moves:

0-63 = move one unit to the left
64-127 = move one unit to the right
128-191 = move one unit up
192-255 = move one unit down

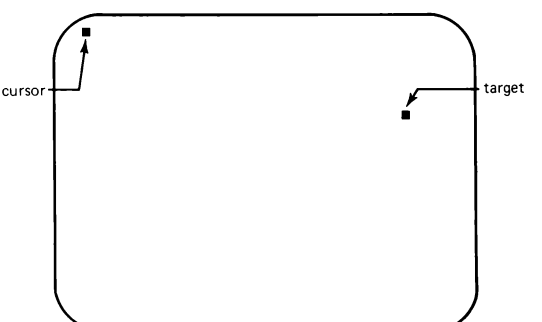


Figure 8-1. Blind Target Shoot.

The actual correspondence between the physical rotation of the paddle and the cursor moves must be determined by practice.

The computer reads the game paddle using the instruction

```
10 PDL(0)
```

The number 0 is the number of the paddle. There may be up to four paddles connected (with numbers 0-3).

There is one further complication. In either of the Franklin's graphics modes, the cursor sits in the text window (the bottom four rows of the screen) and cannot move into the graphics area. Therefore, to use the cursor in a game, it's necessary to create a simulated cursor using a block lit up by the **PLOT** command. To move the simulated cursor, you first turn the cursor off in one location and then turn it on in another.

Here's a sample session with the game. The underlined lines are those you type.

```
RUN

RANDOM NUMBER SEED? 53
BLIND TARGET SHOOT
TO BEGIN GAME, TYPE "READY"
READY
```

The screen clears. The target is displayed. See Figure 8-2.

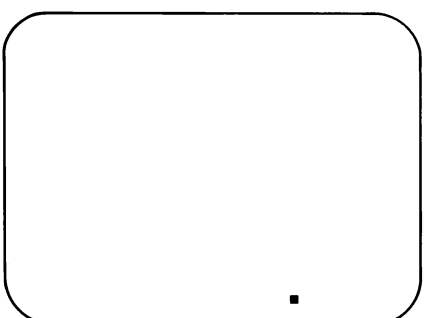


Figure 8-2.

The screen is cleared and the cursor is moved to the home position. See Figure 8-3A. The cursor is then moved to the remembered position of the target. See Figure 8-3B. Time runs out. See Figure 8-3C.

The score is calculated. See Figure 8-4.

Here's a listing of our program.

```
10 PRINT "BLIND TARGET SHOOT"
20 PRINT "TO BEGIN GAME, TYPE READY"
30 INPUT A$
40 IF A$="READY" THEN 90
50 GOTO 10
90 GR
100 GOSUB 2000
110 S=3600*HR+60*MN+SEC
120 Y(0)=INT(40*RND(S)) :REM CHOOSE RANDOM COLUMN
130 X(0)=INT(40*RND(S)) :REM CHOOSE RANDOM ROW
135 COLOR=1
140 PLOT X(0),Y(0):REM DISPLAY TARGET
145 GOSUB 2000
147 T=3600*HR+60*MN+SEC
150 IF T-S=2 THEN 200
160 GOTO 145 : REM WAIT 2 SECONDS
200 COLOR=0 : PLOT X(0),Y(0):REM TURN OFF TARGET
210 COLOR=1
220 PLOT 0,0 :REM TURN CURSOR BACK ON
300 GOSUB 2000
305 S=3600*HR+60*MN+SEC
310 X=1:Y=1: REM (X,Y) ARE COORDINATES OF CURSOR
400 A=PDL(0): REM READ GAME PADDLE
500 IF A>=128 AND A<=191 THEN 510:REM CURSOR UP
505 GOTO 600
510 IF Y>0 THEN 520
515 GOTO 1000
520 COLOR=0: PLOT X,Y: COLOR=1
530 Y=Y-1: PLOT X,Y
540 GOTO 1000
600 IF A>=192 AND A<=255 THEN 610:REM CURSOR DOWN
605 GOTO 700
610 IF Y<39 THEN 620
615 GOTO 1000
620 COLOR=0 :PLOT X,Y:COLOR=1
630 Y=Y+1: PLOT X,Y
640 GOTO 1000
700 IF A>=64 AND A<=127 THEN 710: REM CURSOR RIGHT
705 GOTO 800
710 IF Y<39 THEN 720
```

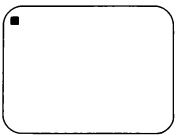


Figure 8-3A.

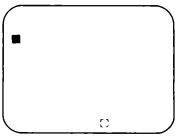


Figure 8-3B.

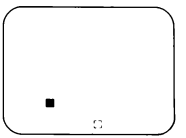


Figure 8-3C.

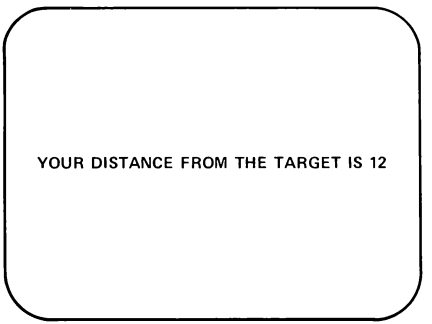


Figure 8-4.

```
715 GOTO 1000
720 COLOR=0: PLOT X,Y:COLOR=1
730 Y=Y+1:PLOT X,Y
740 GOTO 1000
800 IF A<=83 THEN 810:REM CURSOR LEFT
805 GOTO 1000
810 IF X>0 THEN 820
815 GOTO 1000
820 COLOR=0:PLOT X,Y:COLOR=1
830 X=X-1:PLOT X,Y
840 GOTO 1000
1000 REM IS TIME UP?
1010 GOSUB 2000
1015 T=3600*HR+60*MN+SEC
1020 IF T-S=10 THEN 1100
1025 GOTO 400
1100 T=ABS(X-X(0))+ABS(Y-Y(0)):REM T=DIST. TO TARGET
1105 HOME
```

```
1110 PRINT "YOUR DISTANCE FROM THE TARGET IS",T
1120 IF T=0 THEN PRINT "CONGRATULATIONS!"
1130 IF T=0 PRINT "YOU HIT THE TARGET."
1140 SC=100
1150 IF T>0 THEN SC=SC-10
1160 IF T>2 THEN SC=SC-20
1170 IF T>5 THEN SC=SC-20
1180 IF T>10 THEN SC=SC-20
1190 IF T>15 THEN SC=SC-20
1200 IF T>20 THEN SC=SC-10
1300 PRINT "YOUR SCORE IS ", SC
1400 INPUT "DO YOU WISH TO PLAY AGAIN(Y/N)";B$
1410 IF B$="Y" THEN 90
1500 END
2000 PR#3:IN#3:REM READ CLOCK
2010 INPUT "#"; MO,DW,DT,HR,MN,SEC
2020 PR#0:IN#0
2030 RETURN
```

Exercises

1. Experiment with the above program by making the time of target viewing shorter or longer than one second.
2. Experiment with the above program by making the time for target location shorter or longer than 10 seconds.
3. Modify the program to keep a running total score for a sequence of 10 games.
4. Modify the program to allow two players, keeping a running total score for a sequence of 10 games. At the end of ten games, the computer should announce the total scores and declare the winner.

8.3 Tic Tac Toe

In this section, you'll find a program for the traditional game of tic tac toe. The computer won't execute a strategy. Rather, it will be fairly stupid and choose its moves randomly. You'll use the random number generator to "flip" for the first move. Throughout the program, you'll be "O" and the computer will be "X." Here's a sample game.

TEST YOUR UNDERSTANDING 1 (answer on page 159)

How can the computer toss to see who goes first?

The computer now draws a TIC TAC TOE board. See Figure 8-7.

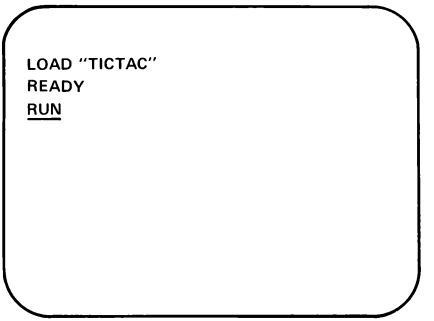


Figure 8-5.

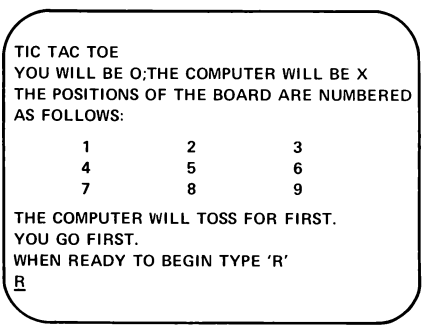


Figure 8-6.

The computer now displays your move and makes a move of its own. See Figure 8-8.

The computer will now make its move and so on until someone wins or a tie game results.

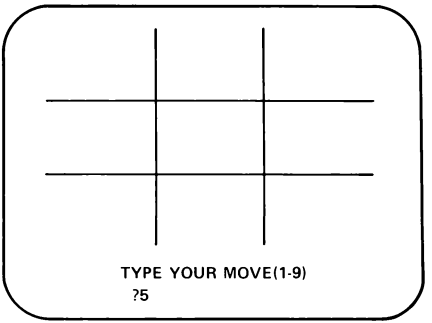


Figure 8-7.

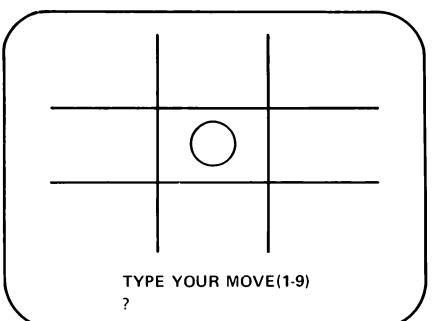


Figure 8-8.

The program below uses much of what you've learned.

First of all, you've used the graphics mode to draw the TIC TAC TOE board, beginning in line 2000. You've structured the program so that it consists of a series of subroutines. Lines 2000-2999 contain the instructions to draw the board and to display the current status of the game. Lines 3000-3999 contain a subroutine which inputs your move. Lines 4000-4999 contain a

subroutine which lets the computer decide its move. Lines 5000-5999 process the current move and decide if the game is over. The logic in lines 6000-8000 allows the computer to determine if there is a winning move or if it should block its opponent.

Here are the variables used in the program.

Z = 0 if it's your move and Z = 1 if it is the computer's.
A$(J) (J = 1,2,...,9) contain either O, X, or the empty string, indicating the current status of position J.
S = the position of the current move.
M = the number of moves played (including the current one).

A video display worksheet was used to lay out the board, and to determine the coordinates for the lines and the X's and O's.

Here's a listing of our program.

```
10 REM ************************
20 REM *      TIC TAC TOE      *
30 REM ************************
40 REM  FRANKLIN LIKES 1 FOR RND
50 LET Q = 1
60 TEXT
70 HOME
80 LET W=0
90 DIM A$(9)
100 DIM B$(3,3,3)
110 FOR I=1 TO 9:A$(I)="": NEXT I
120 HOME : VTAB 6: HTAB 12
130 PRINT "TIC TAC TOE"
140 PRINT
150 PRINT "YOU WILL BE O; THE COMPUTER WILL BE X."
160 PRINT
170 PRINT "THE POSITIONS ON THE BOARD ARE NUMBERED"
180 PRINT
190 PRINT "AS FOLLOWS:": PRINT
200 PRINT "          1,2,3"
210 PRINT "          4,5,6"
220 PRINT "          7,8,9"
230 FOR I=0 TO 3000: NEXT I: PRINT :I=0: REM  SHORT DELAY
240 PRINT "THE COMPUTER WILL TOSS FOR FIRST."
250 FOR ZZ=0 TO 1500: NEXT ZZ: REM TIME DELAY FOR TOSS
260 IF RND (Q) >.5 THEN 280
270 GOTO 310
280 PRINT : PRINT "YOU GO FIRST !"
290 Z=0
300 GOTO 330
310 PRINT : PRINT "I'LL GO FIRST !"
```

```
320 Z=1
330 PRINT : PRINT "WHEN READY TO BEGIN, TYPE 'R'."
340 GET C$
350 IF C$="R" THEN 370
360 GOTO 330
370 GOSUB 490
380 FOR M=1 TO 9: REM  M=MOVE#
390 IF Z=0 THEN  GOSUB 1170
400 IF Z=1 THEN  GOSUB 1260
410 Z=1-Z: REM  Z=1 MEANS NEXT MOVE IS COMPUTER'S
420 IF W=1 THEN 460
430 NEXT M
440 REM TIE GAME
450 PRINT "THE GAME IS TIED."
460 PRINT " BYE, IT'S BEEN FUN !"
470 REM LOOKS LIKE GAMES OVER
480 END
490 REM  DRAW TIC TAC TOE BOARD
500 GR : COLOR=15: REM  SET GRAPHICS,COLOR 15 IS WHITE
510 FOR J=1 TO 2
520 FOR K=0 TO 39
530 PLOT K,13*J
540 NEXT K
550 NEXT J
560 FOR J=1 TO 2
570 FOR K=0 TO 39
580 PLOT 13*J,K
590 NEXT K
600 NEXT J
610 REM  DISPLAY CURRENT GAME STATUS
620 A=7:B=7
630 IF A$(1)="X" THEN  GOSUB 900
640 IF A$(1)="O" THEN  GOSUB 1020
650 A=20:B=7
660 IF A$(2)="X" THEN  GOSUB 900
670 IF A$(2)="O" THEN  GOSUB 1020
680 A=33:B=7
690 IF A$(3)="X" THEN  GOSUB 900
700 IF A$(3)="O" THEN  GOSUB 1020
710 A=7:B=20
720 IF A$(4)="X" THEN  GOSUB 900
730 IF A$(4)="O" THEN  GOSUB 1020
740 A=20:B=20
750 IF A$(5)="X" THEN  GOSUB 900
760 IF A$(5)="O" THEN  GOSUB 1020
770 A=33:B=20
780 IF A$(6)="X" THEN  GOSUB 900
```

```
790 IF A$(6)="O" THEN  GOSUB 1020
800 A=7:B=33
810 IF A$(7)="X" THEN  GOSUB 900
820 IF A$(7)="O" THEN  GOSUB 1020
830 A=20:B=33
840 IF A$(8)="X" THEN  GOSUB 900
850 IF A$(8)="O" THEN  GOSUB 1020
860 A=33:B=33
870 IF A$(9)="X" THEN  GOSUB 900
880 IF A$(9)="O" THEN  GOSUB 1020
890 RETURN
900 REM  DRAW X
910 COLOR=2: REM   BLUE
920 PLOT A,B: REM  CENTERED ON A,B
930 PLOT A-1,B-1
940 PLOT A-1,B+1
950 PLOT A-2,B-2
960 PLOT A-2,B+2
970 PLOT A+1,B+1
980 PLOT A+1,B-1
990 PLOT A+2,B+2
1000 PLOT A+2,B-2
1010 RETURN
1020 REM  DRAW O
1030 COLOR=13: REM  YELLOW
1040 PLOT A-2,B: REM  CENTERED AT A,B
1050 PLOT A-2,B+1
1060 PLOT A-2,B-1
1070 PLOT A+2,B
1080 PLOT A+2,B+1
1090 PLOT A+2,B-1
1100 PLOT A,B-2
1110 PLOT A-1,B-2
1120 PLOT A+1,B-2
1130 PLOT A,B+2
1140 PLOT A-1,B+2
1150 PLOT A+1,B+2
1160 RETURN
1170 PRINT "TYPE YOUR MOVE (1-9) "
1180 GET S$:S=VAL (S$)
1190 IF S < 1 OR S > 9 THEN 1170
1200 IF A$(S)="X" OR A$(S)="O" THEN PRINT "ILLEGAL MOVE ":
     GOTO 1170
1210 A$(S)="O": REM  MAKES MOVE
1220 GOSUB 490: REM  DISPLAY IT
1230 C$="O"
1240 GOSUB 1350: REM  GAME OVER?
```

```
1250 RETURN
1260 GOTO 1510: REM   WIN OR BLOCK ?
1270 S=INT ( RND (0)*9+1): REM  NO, ANY MOVE WILL DO
1280 IF A$(S)="" THEN 1300: REM  EMPTY?
1290 GOTO 1260: REM  NO,TRY AGAIN
1300 A$(S)="X": REM  THE MOVE!
1310 GOSUB 490: REM  DISPLAY IT
1320 C$="X": REM  COMPUTER'S PLAY
1330 GOSUB 1350: REM  GAME OVER ?
1340 RETURN : REM  NEXT MOVE
1350 REM  IS GAME OVER ?
1360 REM
1370 IF A$(1)=A$(2) AND A$(2)=A$(3) AND A$(3)=C$ THEN 1460
1380 IF A$(1)=A$(4) AND A$(4)=A$(7) AND A$(7)=C$ THEN 1460
1390 IF A$(1)=A$(5) AND A$(5)=A$(9) AND A$(9)=C$ THEN 1460
1400 IF A$(2)=A$(5) AND A$(5)=A$(8) AND A$(8)=C$ THEN 1460
1410 IF A$(3)=A$(6) AND A$(6)=A$(9) AND A$(9)=C$ THEN 1460
1420 IF A$(4)=A$(5) AND A$(5)=A$(6) AND A$(6)=C$ THEN 1460
1430 IF A$(7)=A$(8) AND A$(8)=A$(9) AND A$(9)=C$ THEN 1460
1440 IF A$(3)=A$(5) AND A$(5)=A$(7) AND A$(7)=C$ THEN 1460
1450 GOTO 1500
1460 REM LOOKS LIKE A WINNER
1470 PRINT "LOOKS LIKE ";C$;" WINS THIS GAME"
1480 LET W=1
1490 REM  END OF GAME
1500 RETURN
1510 REM  NOW LOOK FOR WINNING MOVE
1520 REM  OR A BLOCKING MOVE
1530 COUNT=0
1540 FOR I=1 TO 9
1550 IF A$(I)="X" THEN B(I)=-1
1560 IF A$(I)="" THEN B(I)=0
1570 IF A$(I)="O" THEN B(I)=1
1580 NEXT I
1590 REM CONVERTS DISREGARDING ORDER
1600 READ I,J,K
1610 COUNT=COUNT+1
1620 IF COUNT > 8 THEN 1730
1630 REM  ALL CHECKED ?
1640 S1=B(I)+B(J)+B(K)
1650 IF S1=-2 THEN 1690
1660 REM  TWO X'S IN A ROW ?
1670 GOTO 1600
1680 REM NO, KEEP ON LOOKING
1690 IF B(J)=0 THEN A$(J)="X": GOTO 1840
1700 IF B(K)=0 THEN A$(K)="X": GOTO 1840
1710 IF B(I)=0 THEN A$(I)="X": GOTO 1840
```

```
1720 REM  TAKE IT AND RUN
1730 RESTORE : REM   LOOK FOR BLOCK
1740 COUNT=0
1750 READ I,J,K
1760 COUNT=COUNT+1
1770 IF COUNT > 8 THEN 1850
1780 S1-B(I)+B(J)+B(K): REM  WATCH OUT FOR THIS ONE!
1790 IF S1=2 THEN 1810
1800 GOTO 1750
1810 IF B(J)=0 THEN A$(J)="X": GOTO 1840
1820 IF B(K)=0 THEN A$(K)="X": GOTO 1840
1830 IF B(I)=0 THEN A$(I)="X": GOTO 1840
1840 RESTORE : GOTO 1310: REM  WIN OR BLOCK;DISPLAY IT
1850 RESTORE : GOTO 1270: REM  ANY MOVE WILL DO
1860 DATA 1,2,3,4,5,6,7,8,9,1,4,7,2
1870 DATA 5,8,3,6,9,1,5,9,3,5,7,1,2,3
1880 REM  ***** WARNING *****
1890 REM  DO NOT OMIT THE NUMBERS 1,2,3 IN LINE 1870.
```

Exercises

1. Modify the above program so that you and the computer may play a series of ten games. The computer should decide the champion of the series.
2. Modify the above program to play 4x4 Tic Tac Toe.

ANSWER TO TEST YOUR UNDERSTANDING 1

See lines 120-170 of the program on page 155.

9

Programming for Scientists

In this chapter you'll discover some aspects of Franklin computer programming of interest to scientists, engineers, and mathematicians. In particular, you'll find the library of mathematical functions which can be used in Floating Point BASIC.

9.1 Integer and Real Constants

Up to this point, you've used the computer to perform arithmetic without giving much thought to the level of accuracy of the numbers involved. However, when doing scientific programming, it's absolutely essential to know the number of decimal places of accuracy of the computations. Let's begin this chapter by discussing the form in which Floating Point BASIC stores and utilizes numbers.

Floating Point BASIC recognizes two different types of numeric constants: integer and real.

An **integer** numeric constant is an ordinary integer (positive or negative) in the range -32768 to + 32767. (32768 is 2 raised to the 15th power. This number is significant for the internal workings of the Franklin computer.) Here are some examples of integer numeric constants:

7, 58, 3712, -15, -598

Integer numeric constants may be stored very efficiently in RAM. Therefore, in order to realize these efficiencies, the Franklin recognizes integer numeric constants and handles them in a special way.

A **real constant** is a number having nine or fewer digits. Some examples of real constants are

5.135, -63.5785, 1234567, -1.467654E12

Note that a real constant may be expressed in "scientific" or "floating point" notation, as in the final example shown here. In such an expression, however, you're limited to nine or fewer digits. In Floating Point BASIC, real constants must lie within these ranges: Between $-1x10^{38}$ and $-1x10^{-38}$; Between $1x10^{-38}$ and $1x10^{38}$. This is a limitation that's seldom much of a limitation in practice. After all, $1x10^{-38}$ equals

.00000000000000000000000000000000000001

(37 zeros followed by a 1), which is about as small a number as you're ever likely to encounter! Similarly, $1x10^{38}$ equals

100,000,000,000,000,000,000,000,000,000,000,000

(a 1 followed by 38 zeros), which is large enough for most practical calculations.

Real constants occupy more RAM than do integer constants. Moreover, arithmetic with real constants proceeds slower than integer arithmetic. The Franklin recognizes both types of numerical constants, and uses only as much arithmetic power as necessary.

Here are the rules for determining the type of a numerical constant:

1. Any integer in the range -32767 and 32767 is an integer constant.
2. Any number having nine or fewer digits and not an integer constant is a real constant. Any number in scientific notation using E before the exponent is assumed to be a real constant. If a number has more than nine digits, it will be truncated after the ninth digit. For example, the number

   1.234567891234E15

   will be interpreted as the real constant

   1.23456789E15.

For arithmetic operations, the Franklin converts all constants to real constants.

It's important to realize that if a number does not have an exact decimal representation (such as 1/3 = .333...) or if the number has a decimal representation which has too many digits for the constant type being used, the computer will then be working with an approximation of the number rather than the number itself. The built-in errors caused by the approximations of the computer are called **round-off errors**. Consider the problem of calculating

```
1/3+1/3+1/3
```

As seen above, 1/3 is stored as the real constant

```
.333333333+.333333333+.333333333=
```

The sum has a round-off error of .000000001.

The Franklin computer displays up to nine digits for a real constant. However, due to round-off error, the answer to any single arithmetic operation is guaranteed accurate to only eight places.

Exercises (answers on page 213)

For each of the constants below, determine the number stored by the computer.

1. 3
2. 2.37
3. 5.78E5
4. 2
5. 3
6. -4.1
7. -4.1
8. 3500.6847586958658
9. 2.176E2
10. -5.94E12
11. 3.586950400383726537 4
12. -2345423837 46.21
13. -2.3b7E20
14. 4570000000000000000

For each of the arithmetic problems below, determine the number as stored by the computer.

15. 1+45
16. 2/4
17. 3/5
18. 3/5+1
19. 2/3
20. 2/3+.53
21. .5E4 -.37E2
22. 1.75E3-1.0E-5

23. Calculate $1/3 + 1/3 + 1/3 + \ldots + 1/3$ (10 1/3's) using real constants. What answer is displayed? Is this answer accurate to eight digits? If not, explain why.

9.2 Variable Types

In the previous section you were introduced to the types of numerical constants: integer and real. There is a parallel set of types for variables.

A variable of integer type takes on values which are integer type constants. An integer type variable is indicated by the symbol % after the variable name. For example, here are some variables of integer type:

```
A%, BB%, A1%
```

In setting the value of an integer type variable, the computer will truncate any fractional parts to obtain an integer. For example, the instruction

```
10 LET A%=2.54
```

will set the value of A equal to the integer constant 2. Integer type variables are useful when keeping track of integer quantities, such as line numbers in a program.

A variable of real type is one whose value is a real constant. A **real type variable** is any variable not an integer or string type. Here are some examples of real variables:

```
K, W7, ZX
```

In setting the value of a real variable, all digits beyond the ninth are truncated. For example, the instruction

```
20 LET A=1.23456789123
```

will set A equal to 1.23456789.

If a variable is used without a type designator, then the computer will assume that it is a real variable. All of the variables used until now have been real variables. These are, by far, the most commonly used variables.

Note that the variables A%, A, and A$ are three distinct variables. You could, if you wish, use all of them in a single program.

TEST YOUR UNDERSTANDING 1 (answers on page 165)

What values are assigned to each of these variables?

a. A = 1
b. C% = 5.22
c. BB = 1387.56999999

Exercises (answers on page 213)

Calculate the following quantities in real arithmetic.

1. (5.87 + 3.85-12.07)/11.98
2. (15.1 + 11.9)^4/12.88
3. (32485 + 9826)/(321.5-87.6^2)
4. Write a program to determine the largest integer less than or equal to X, where the value of X is supplied in an **INPUT** statement.

Determine the value assigned to the variable in each of the following exercises.

5. A% = -5
6. A% = 4.8
7. A% = -11.2
8. A = 1.78
9. A = .001
10. A = 32.653426278374645237
11. A = 4.25234544321E21
12. A = -1.23456789E-32
13. A = 3.283646493029273646434
14. A = -5.74

ANSWERS TO TEST YOUR UNDERSTANDING 1

a. 1.00000000
b. 5
c. 1387.56999

9.3 Mathematical Functions in Floating Point BASIC

In performing scientific computations, it's often necessary to use a wide variety of mathematical functions, including the natural logarithm, the exponential, and the trigonometric functions. The Franklin computer has a wide range of these functions "built-in." This section describes these functions and their use.

All mathematical functions in Floating Point BASIC work in a similar fashion. Each function is identified by a sequence of letters (**SIN** for sine, **LOG** for natural logarithm, and so forth). To evaluate a function at a number X, write X in parentheses after the function name. For example, the natural logarithm of X is written **LOG(X)**. The program will use the current value of the variable X and will evaluate the natural logarithm of that value. For example, if X is currently 2, then the computer will calculate **LOG(2)**.

Instead of the variable X, you may use any type of variable: integer or real. You may also use a numerical constant of any type. For example, **SIN**(.435678889658595) asks for the sine of a real numerical constant. Note that with only a few exceptions (see below), all Floating Point BASIC functions return a real result, accurate to nine digits. For example, the above value of the sine function will be computed as

```
SIN(.435678889658595)=.422025969
```

Floating Point BASIC lets you evaluate a function at any expression. Consider the expression X^2 + Y^2-3*X. It's perfectly acceptable to call for calculations such as

```
SIN(X^2+Y^2-3*X)
```

The computer will first evaluate the expression X^2 + Y^2-3*X using the current values of the variables X and Y. For example, if X = 1 and Y = 4, then X^2 + Y^2-3*X = 12 + 42-3*1 = 14. So the above sine function will be evaluated as **SIN**(14) = .990607356.

Trigonometric Functions

The Franklin has the following trigonometric functions available.

SIN(X) = the sine of the angle X
COS(X) = the cosine of the angle X
TAN(X) = the tangent of the angle X

Here the angle X is expressed in terms of radian measure. In this measurement system, 360 degrees equals 2 radians. Or one degree equals .17453 radians and one radian equals 57.29578 degrees. If you want to calculate trigonometric functions with the angle X expressed in degrees, use these functions:

```
SIN (.017453*X)
COS(.017453*X)
TAN(.017453*X)
```

The three other trigonometric functions, **SEC(X)**, **CSC(X)**, and **COT(X)**, may be computed from the formulas

```
SEC(X)=1/COS(X)
CSC(X)=1/SIN(X)
COT(X)=COS(X)/SIN(X)
```

Here, as above, the angle X is in radians. To compute these trigonometric functions with the angle in degrees, replace X by

```
.017453*X
```

Floating Point BASIC has only one of the inverse trigonometric functions, namely the arctangent, denoted **ATN(X)**. This function returns the angle whose tangent is X. The angle returned is expressed in radians. To compute the arctangent with the angle expressed in degrees, use the function

```
57.29578*ATN(X)
```

TEST YOUR UNDERSTANDING 1 (answer on page 169)

Write a program which calculates sin 45°, cos 45° and tan 45°.

Logarithmic and Exponential Functions

Floating Point BASIC allows you to compute e^x using the exponential function:

```
EXP(X)
```

Furthermore, you may compute the natural logarithm of X via the function

```
LOG(X)
```

You may calculate logarithms to base b using the formula:

```
LOG,(X)=LOG(X)/LOG(b)
```

Example 1. Prepare a table of values of the natural logarithm function for values X = .01, .02, .03, ..., 100.00. Output the table on the printer.

Solution. Here's the desired program. Note that you've prepared your table in two columns with a heading over each column.

```
5 PR#1
10 PRINT "X", "LOG(X)"
20 J=.01
30 PRINT J, LOG(J)
40 IF J=100.00 THEN 100
45 GOTO 50
50 J=J+.01
60 GOTO 30
70 PR#0
100 END
```

TEST YOUR UNDERSTANDING 2 (answer on page 169)

Write a program which evaluates the function $f(x) = \sin x / (\log x + e^x)$ for x = .45 and x = .7.

Example 2. Carbon dating is a technique for calculating the age of ancient artifacts by measuring the amount of radioactive carbon-14 remaining in the artifact, as compared with the amount present if the artifact were manufactured today. If r denotes the proportion of carbon-14 remaining, then the age (A) of the object is calculated from the formula:

A = -(1/.00012)*LOG(r)

Suppose that a papyrus scroll contains 47% of the carbon-14 of a piece of papyrus just manufactured. Calculate the age of the scroll.

Solution. Here r = .47. So we use the above formula:

```
10 LET R=.47
20 LET A=-(1/.00012)*LOG(R)
30 PRINT "THE AGE OF THE PAPYRUS IS", A, "YEARS"
40 END
```

Powers

Floating Point BASIC has a square root function, denoted **SQR(X).** As with all the functions considered so far, this function will accept as input an integer or a real constant and will output a real constant. For example, the instruction

```
10 LET Y=SQR(2)
```

will set Y equal to 1.41421356.

Actually, the exponentiation procedure you learned in Chapter 2 will work equally well for fractional and decimal exponents, and therefore provides an alternative method for extracting square roots. Here's how to use it. Taking the square root of a number corresponds to raising the number to the 1/2 power. You may calculate the square root of X as

```
X^(1/2)
```

Note that the square root function **SQR(X)** operates with greater speed and is therefore preferred. The alternate method is more flexible, however. For instance, you may extract the cube root of X as

```
X^(1/3)
```

or raise X to the 5.389 power as follows:

```
X^5.389
```

Greatest Integer, Absolute Value, and Related Functions

Here are several extremely helpful functions. The greatest integer less than or equal to X is denoted **INT(X).** For example, the largest integer less than or equal to 5.46789 is 5, so

```
INT(5.46789)=5
```

Similarly, the largest integer less than or equal to -3.4 is -4 (on the number line, -4 is the first integer to the left of -3.4). Therefore:

```
INT(-3.4)=-4
```

Note that the **INT** function throws away the decimal part of a positive number, although this description does not apply to negative numbers.

The absolute value of X is denoted **ABS(X).** Recall that the absolute value of X is X itself if X is positive, or 0 and equals -X if X is negative. Thus:

```
ABS(9.23)=9.23
   ABS(0)=0
ABS(-4.1)=4.1
```

Just as the absolute value of X "removes the sign" of X, the function **SGN(X)** throws away the number and leaves only the sign. For example:

```
SGN(3.4)=+1
SGN(-5.62)=-1
SGN(0)=0
```

Exercises (answers on page 214)

Calculate the following quantities:

1. $e^{1.54}$
2. $e^{-2.376}$
3. log 58
4. log .0000975
5. sin 3.7
6. cos 45°
7. arctan 1
8. tan .682
9. arctan 2 (expressed in degrees)
10. $\log_{10} 18.9$
11. Make a table of values of the exponential function exp(x) for x = −5.0, −4.9, ... , 0 , .1, ... , 5.0.
12. Evaluate the function

    ```
    3x^(1/4)log(5x)+exp(-1.8x)tan x
    ```

    for x = 1.7, 3.1, 5.9, 7.8, 8.4, and 10.1.
13. Write a Floating Point BASIC program to graph the function y = sin x for x from 0 to 6.28. Use an interval of .05 on the x axis.
14. Write a Floating Point BASIC program to graph the function y = ABS(x).
15. Write a program to calculate the fractional part of x. (The fractional part of x is the portion of x which lies to the right of the decimal point.)

ANSWERS TO TEST YOUR UNDERSTANDING 1 and 2

1:
```
10 LET A=.017453
20 PRINT SIN(45*A), COS(45*A), TAN(45*A)
30 END
```

2:
```
10 DATA .45,.7
20 FOR J=1 TO 2
30 READ A(J)
40 PRINT SIN(A(J))/(LOG(A(J))+EXP(A(J)))
50 NEXT J
60 END
```

9.4 Defining Your Own Functions

In mathematics, functions are usually defined by specifying one or more formulas. For instance, here are formulas which define three functions f(x), g(x), and h(x):

$f(x) = (x^2-1)^{1/2}$

$g(x) = 3x^2-5x-15$

$h(x) = 1/(x-1)$

Note that each function is named by a letter, namely f, g, and h, respectively. Floating Point BASIC allows you to define functions like these and to use them by name throughout your program. To define a function, you use the **DEF FN** instruction. This instruction is used before the first use of the function in the program. For example, to define the function f(x) above, you could use the instruction

```
10 DEF FNF(X)=(X^2-1)^(1/2)
```

To define the function g(x) above, use the instruction

```
20 DEF FNG(X)=3*X^2-5*X-15
```

Note that in each case, you use a letter (F or G) to identify the function. Suppose that you wish to calculate the value of the function F for X = 12.5. Once the function has been defined, this calculation may be described to the computer as FNF(12.5). Such calculations may be used throughout the program and save the effort of retyping the formula for the function in each instance. You may use any valid variable name as a function name. For example, you may define a function INTEREST by the statement:

```
10 DEF FNINTEREST(X)=...
```

Note, however, that only the first two letters of the variable are used to identify the function. Any other variable name beginning with **IN** would refer to the same function as the one above.

In defining a function, you may use other functions. For example, if FNF(X) and FNG(X) are as defined above, then you may define their product by the instruction

```
30 DEF FNC(X)=FNF(X)*FNG(X)
```

Exercises (answers on page 215)

Write instructions to define the following functions.

1. x^2-5x
2. $1/x-3x$
3. $5\exp(-2x)$
4. $x\log(x/2)$

5. tan x/x
6. cos(2x) + 1
7. Write a program which tabulates the value of the function in Exercise 3 for x = 0, .1, .2, .3, .4, ... , 10.0.

10 Computer-Generated Simulations

10.1 Simulation

Simulation is a powerful analysis tool which lets you use your computer to perform experiments to solve a wide variety of problems which might be too difficult to solve otherwise.

To describe what simulation is, let's use a concrete example. Assume that you own a dry cleaning store. At the moment, you have only one salesperson behind the counter, but you are considering adding a second. Your question is: Should you hire the extra person? Being an analytical person, you've collected the following data. Traffic through your store varies by the hour. However, you have kept a log for the past month and are able to estimate the average number of potential customers arriving in the shop according to the following table:

7-8 AM	30	1-2 PM	9
8-9	15	2-3	8
9-10	6	3-4	12
10-11	3	4-5	12
11-12	8	5-6	35
12-1 PM	25	6-7	22

You've observed that you are currently paying a penalty for not having a second salesperson: If there is too long a wait, a customer will go somewhere else to have his or her clothes cleaned! In your observations, you've noted that, on the average, of people entering the shop a certain percentage will leave, depending on the size of the line. The likelihood that a person leaves depends on whether it is a drop-off or a pick-up. Those picking up clothes are more likely to wait in line. Here are the results of your observations:

line size	% leaving (drop-off)	% leaving (pick-up)
0	0	0
1-3	15	5
4-6	25	15
7-10	60	35
11-15	80	50

The average time to wait on a person is four minutes and the size of the average cleaning bill is $5.75. The cost of hiring the new salesperson is 200 dollars per week. Assuming that the salespersons work continuously while the shop is open, what action should you take?

This problem is fairly typical of the problems which arise in business. It's characterized by data accumulated from observations and unpredictable events. (When will a given customer arrive? Will he or she encounter a long line? Will he or she be the impatient sort who walks out?) Nevertheless, you must make a decision based on the data you have. How should you proceed?

One technique is to let your computer "imitate" your shop. Let the computer play a game which consists of generating customers at random times. These customers enter the "shop" and, on the basis of the current line, decide whether or not to stay. The computer will keep track of the line, the number of customers who leave, the revenue generated, and the revenue lost. The computer will keep up the simulated traffic for an entire "day" and present you with the results of the daily activity. But, you might argue, the computer data may not be valid. Suppose that it generates a "non-typical" day. Its data might be biased. This could, indeed, happen. In order to avoid this pitfall, we run the program for many simulated days and average the results. The process just described is called **simulation**.

In this chapter you'll get a glimpse of the power of simulation and come away with enough of an idea to build simple simulations of your own.

First, let's handle some of the mathematical ideas needed for the next section. The required notions center around the following question: How do we make the computer imitate an unpredictable event? Consider the irate customer who arrives to drop off cleaning and encounters a line of four people ahead of him. According to the above table, the customer will leave 25 percent of the time and remain in line 75 percent of the time. How do you let the computer make the decision for the customer?

Easy! Use the random number generator. Recall that RND(1) generates a random number between 0 (included) and 1 (excluded). Suppose you ask how often RND(1) is larger than .25. If, indeed, the numbers produced by the random number generator show no biases, approximately 75 percent of the numbers produced will lie in the given interval since this interval occupies 75 percent of the length of the interval from 0 to 1. Let our customer decide as follows: If RND(1) > .25 then the customer joins the line; otherwise, the

customer walks out in a huff. You'll employ this simple idea several times in designing your simulation.

10.2 Simulation of a Dry Cleaning Store

Let's build a simulation to solve the problem stated in the preceding section by first deciding on techniques for imitating each of the important aspects of the problem.

Since the problem calls for analysis of actions as time passes, you must somehow measure the passage of (simulated) time. To do this, use the variables TH (time-hours) and TM (time-minutes) to keep track of the current time. In order to avoid a problem with AM and PM, let's use the military time system. In this system the PM hours are denoted as 13 through 24. For example, 1:15 PM is shown as 13:15. As the unit of simulated time, let's use four minutes, the time it takes to serve one customer. Your program will then look at time in four-minute segments. During each four-minute segment, it will take certain actions and then advance to the next time segment by adjusting TH and TM. When TM exceeds 60 minutes (an hour), subtract 60 and credit the 60 minutes to TH, which is increased by 1. Let's do the time advance in a subroutine at line 1000. Here's the subroutine.

```
1000 REM TIME ADVANCE
1010 LET TM=TM+4
1020 IF TM >= 60 THEN 1030
1025 GOTO 1100
1030 TM = TM-60: TH=TH+1
1100 RETURN
```

Let's store the statistical data on customer arrivals in the array A(H) (H = 7,8,...,18). A(7) will equal the number of customers arriving between 7 and 8 AM, A(8) the number arriving between 8 and 9 AM,..., A(18), the number arriving between 6 and 7 PM. The first action of the program is to set up this array.

```
10 DIM A(18)
20 DATA 30,15,6,3,8,25,9,8,12,12,35,22
30 FOR H=7 TO 18
40 READ A(H)
50 NEXT H
```

The next step will be to read in the customer "impatience data." Let DROPOFF(K) be the percentage of drop-off customers who leave when the line is K people long. Let PICKUP(K) be the corresponding statistic for pick-up customers. Here's the portion of the program which sets up these arrays.

```
100 DIM DROPOFF(20), PICKUP(20)
110 DATA 0,0,.15,.05,.25,.15,.60,.35,.80,.50
115 READ DROPOFF(0),PICKUP(0)
120 READ DROPOFF(1),PICKUP(1)
130 DROPOFF(2)=DROPOFF(1): DROPOFF(3)=DROPOFF(1)
140 PICKUP(2)=PICKUP(1): PICKUP(3)=PICKUP(1)
150 READ DROPOFF(4), PICKUP(4)
160 DROPOFF(5)=DROPOFF(4): DROPOFF(6)=DROPOFF(4)
170 PICKUP(5)=PICKUP(4): PICKUP(6)=PICKUP(4)
180 READ DROPOFF(7), PICKUP(7)
190 DROPOFF(8)=DROPOFF(7): DROPOFF(9)=DROPOFF(7):
    DROPOFF(10)=DROPOFF(7)
200 PICKUP(8)=PICKUP(7): PICKUP(9)=PICKUP(7):PICKUP(10)=
    PICKUP(7)
210 READ DROPOFF(11), PICKUP(11)
220 FOR J=12 TO 15
230 DROPOFF(J)=DROPOFF(11): PICKUP(J)=PICKUP(11)
240 NEXT J
```

The next step in our program is to set the clock at the beginning of a day. Likewise, the length of the waiting line, indicated by the variable **L**, is set equal to 0, the total of lost cash indicated by the variable **LC**, and the total sales for the day indicated by the variable **CF** (cash flow), are both set equal to 0.

```
300 TH=7: TM=0
310 L=0
320 LC=0
330 CF=0
```

At the beginning of each hour, the program will schedule the arrival of the customers. For the Jth hour, it will schedule the arrival of A(J) customers. Each customer will be given a time of arrival in minutes past the hour. The computer will choose this arrival time using the random number generator. In the absence of any other information, let's assume that the customers spread themselves out in a random, but uniform manner, over the hour. At the beginning of each simulated hour, set up an array D(I) with 15 entries, one for every four-minute period in the hour. This array will indicate how many customers arrive in each four-minute interval. For example, if D(10) = 4, then four customers will arrive between 36 and 40 minutes past the hour (that is, in the tenth four-minute interval in the hour). The program will randomly place each of the A(J) customers in four-minute intervals using the random number generator. The program will test the time for the beginning of an hour. This will be done by determining if TM equals 0 (in line 410). If so, the program will go to a subroutine at 1200 which schedules the arrival of the customers for the hour.

```
11 DIM D(15)
410 IF TM=0 THEN GOSUB 1200
1200 FOR S=1 TO 15
```

```
1210 D(S)=0
1220 NEXT S
1230 FOR I=1 TO A(TH)
1240 X=INT(15*RND(1))+1
1250 D(X)=D(X)+1
1260 NEXT I
1270 RETURN
```

The program now progresses through the simulated hour in four-minute segments. For the Tth four-minute segment, it causes D(T) customers to arrive at the shop. Let's assume, that of these customers, half are drop-off and half are pick-up. The computer lets these customers each look at the line and decide whether to leave or stay. If a customer decides to stay, then he or she is added to the line. If the customer decides to go, the computer makes a note of the $5.75 cash flow lost. The lost cash flow will be stored in the variable LC. After the customers are either in line or have left, the salesperson services a customer (remember, one customer is serviced every four minutes) and $5.75 is added to the cash flow, which is tallied in the variable CF. Finally, the time is updated and the entire procedure is repeated for the next four-minute segment. Let's be rather hard-hearted. If there are any customers left in line at closing time, don't wait on them and add their business to that lost. This rather odd way of doing business is appropriate since you're analyzing the need for more personnel and any overtime should be considered in that analysis. Here is the portion of the program which accomplishes these tasks of the simulation. Note that line 720 tests the time for the end of the working day (TH = 19). When this time arrives, the program goes to line 1500, where the end of day statistics are printed out.

```
420 T=TM/4+1:REM T=# OF 4 MIN. SEGMENTS
430 FOR J=1 TO D(T)
440 LET C=INT(2*RND(1))+1:
445 REM 1=DROP-OFF,2=PICK-UP
450 IF C=2 THEN 600
490 REM 500-560 DOES DROP-OFF CUST. STAY?
500 IF RND(1) > DROPOFF(L) THEN 550
510 LC=LC+5.75: REM CUSTOMER LEAVES
520 GOTO 690
550 L=L+1:REM CUSTOMER JOINS LINE
560 GOTO 690
590 REM 600-660 DOES PICK-UP CUST. STAY?
600 IF RND(1) > PICKUP(L) THEN 640
610 LC=LC+5.75:REM CUSTOMER LEAVES
620 GOTO 690
640 L=L+1:REM CUSTOMER JOINS LINE
690 NEXT J
700 IF L=0 THEN 710 :REM THERE WERE NO CUSTOMERS
701 L=L-1:REM WAIT ON CUSTOMER
705 CF=CF+5.75 : REM TAKE CUSTOMER'S MONEY
```

```
710 GOSUB 1000 : REM UPDATE TIME
720 IF TH=19 THEN 1500
725 GOTO 800
730 REM TH=19 IS THE END OF THE DAY
800 GOTO 410:REM GO TO NEXT 4 MINUTE SEGMENT
1500 PRINT "END OF DAY STATISTICS"
1510 PRINT "BUSINESS LOST", LC+L*5.75
1520 PRINT "CASH FLOW", CF
1530 PRINT "LINE AT CLOSING", L
2000 END
```

The program simulates the activities of a single day. In order to average the statistics over a number of days, let's set up a loop which repeats the above program for a certain number of days. Let's make an arbitrary choice of ten days' repetition. Let the variable D denote the day number. If you also denote the day number by E, then the change of day will be controlled by a loop in lines 290 and 1700.

```
290 FOR E=1 to 10
1700 NEXT E
```

As statistics, let's compute the average of the revenue lost (LC), cash flow (CF), and the line length at closing (L). Keep the totals of these three variables for all the days up to the present in the variables L1, C1, and CL, respectively. You can modify lines 1500-1520 as follows:

```
1500 LET L1=LC+L1+L*5.75
1510 LET C1=CF+C1
1520 LET CL=L+CL
```

Lines 1800-1850 compute the averages of L1, C1, and CL and display the results.

```
1800 LET L1=L1/10
1810 LET C1=C1/10
1820 LET CL=CL/10
1830 PRINT "AVERAGE CASH LOST PER DAY", L1
1840 PRINT "AVERAGE CASH FLOW PER DAY", C1
1850 PRINT "AVERAGE LINE LENGTH AT CLOSING", L
```

This completes the construction of the program. You have carried out the construction of the program in detail so that you could see how a reasonably lengthy program is developed. However, your program is in a rather poor form to read, so let's recopy it in order.

```
10 DIM A(18)
11 DIM D(15)
20 DATA 30,15,6,3,8,25,9,8,12,12,35,22
30 FOR H=7 TO 18
40 READ A(H)
50 NEXT H
```

```
100 DIM DROPOFF(20), PICKUP(20)
110 DATA 0,0,.15,.05,.25,.15,.60,.35,.80,.50
115 READ DROPOFF(0), PICKUP(0)
120 READ DROPOFF(1),PICKUP(1)
130 DROPOFF(2)=DROPOFF(1): DROPOFF(3)=DROPOFF(1)
140 PICKUP(2)=PICKUP(1): PICKUP(3)=PICKUP(1)
150 READ DROPOFF(4), PICKUP(4)
160 DROPOFF(5)=DROPOFF(4): DROPOFF(6)=DROPOFF(4)
170 PICKUP(5)=PICKUP(4): PICKUP(6)=PICKUP(4)
180 READ DROPOFF(7), PICKUP(7)
190 DROPOFF(8)=DROPOFF(7): DROPOFF(9)=DROPOFF(7):
    DROPOFF(10)=DROPOFF(7)
200 PICKUP(8)=PICKUP(7): PICKUP(9)=PICKUP(7):PICKUP(10)=
    PICKUP(7)
210 READ DROPOFF(11), PICKUP(11)
220 FOR J=12 TO 15
230 DROPOFF(J)=DROPOFF(11): PICKUP(J)=PICKUP(11)
240 NEXT J
290 FOR E=1 TO 10
300 TH=7: TM=0
310 L=0
320 LC=0
330 CF=0
410 IF TM=0 THEN GOSUB 1200
420 T=TM/4+1
430 FOR J=1 TO D(T)
440 LET C=INT(2*RND(1))+1:REM 1=DROP-OFF, 2=PICK-UP
450 IF C=2 THEN 600
490 REM 500-560 DOES DROP-OFF CUST. STAY?
500 IF RND(1) > DROPOFF(L) THEN 550
510 LC=LC+5.75: REM CUSTOMER LEAVES
520 GOTO 590
550 L=L+1:REM CUSTOMER JOINS LINE
560 GOTO 590
590 REM 600-660 DOES PICK-UP CUST. STAY?
600 IF RND(1) > PICKUP(L) THEN 640
610 LC=LC+5.75:REM CUSTOMER LEAVES
620 GOTO 690
640 L=L+1:REM CUSTOMER JOINS LINE
690 NEXT J
700 IF L=0 THEN 710 : GOTO 701:REM THERE WERE NO CUSTOMERS
701 L=L-1:REM WAIT ON CUSTOMER
705 CF=CF+5.75 : REM TAKE CUSTOMER'S MONEY
710 GOSUB 1000 : REM UPDATE TIME
720 IF TH=19 THEN 1500
730 REM TH=19 IS THE END OF THE DAY
800 GOTO 410:REM GO TO NEXT 4 MINUTE SEGMENT
```

```
1000 REM TIME ADVANCE
1010 LET TM=TM+4
1020 IF TM < 60 THEN RETURN
1030 TM=TM-60: TH=TH+1
1100 RETURN
1200 FOR S=1 TO 15
1210 D(S)=0
1220 NEXT S
1230 FOR I=1 TO A(TH)
1240 X=INT(15*RND(1))+1
1250 D(X)=D(X)+1
1260 NEXT I
1270 RETURN
1500 LET L1=LC+L1+L*5.75
1510 LET C1=CF+C1
1520 LET CL=L+CL
1700 NEXT E
1800 LET L1=L1/10
1810 LET C1=C1/10
1820 LET CL=CL/10
1830 PRINT "AVERAGE CASH LOST PER DAY "; L1
1840 PRINT "AVERAGE CASH FLOW PER DAY ";C1
1850 PRINT "AVERAGE LINE LENGTH AT CLOSING "; L
2000 END
```

In order to see what's happening at your hypothetical dry cleaning establishment, run your program. Below are the results of five program runs.

```
RUN #1
AVERAGE CASH LOST PER DAY      258.75
AVERAGE CASH FLOW PER DAY      805.00
AVERAGE LINE AT CLOSING             9

RUN #2
AVERAGE CASH LOST PER DAY      270.25
AVERAGE CASH FLOW PER DAY      793.50
AVERAGE LINE AT CLOSING             3

RUN #3
AVERAGE CASH LOST PER DAY      264.50
AVERAGE CASH FLOW PER DAY      799.25
AVERAGE LINE AT CLOSING             4

RUN #4
AVERAGE CASH LOST PER DAY      270.83
AVERAGE CASH FLOW PER DAY      792.93
AVERAGE LINE AT CLOSING             6
```

```
RUN #5
AVERAGE CASH LOST PER DAY      287.50
AVERAGE CASH FLOW PER DAY      776.25
AVERAGE LINE AT CLOSING             4
```

There are several interesting facts about the output. First, note that the runs are not all identical. This is because the random number generator generates new random customer arrival patterns for each run. Second, note the small percentage error in the data from the various runs. There seems to be a statistical pattern which persists from run to run.

Finally, and most significantly, note that you're losing several hundred dollars per day in business because of your inability to service customers. At 200 dollars per week, the additional salesperson is a bargain! Even a single day's lost sales is enough to pay the salary. It appears that you should add the extra salesperson. Actually, a bit more caution is advisable. You were dealing with cash flow rather than profit. In order to make a final decision, you must compute the profit generated by the additional sales. For example, if your profit margin on plant costs (exclusive of sales) is 40 percent, then the profit generated by the extra sales will clearly amount to more than 200 dollars per week and the extra salesperson should be hired.

The above example is fairly typical of the way in which simulation may be applied to analyze even fairly complicated situations in a small business. You'll encounter some further refinements in the exercises.

Exercises

1. Run the above program for ten consecutive runs and record the data. Does your data come close to the data presented above?
2. Suppose that customers become more impatient and the likelihood of leaving is doubled in each case. Rerun the experiment to determine the lost cash flow in this case.
3. Suppose that customers become more patient and the likelihood of leaving is cut in half in each case. Redo the experiment to determine the lost cash flow in this case.
4. Consider the original set of experimental data. Now assume that the second salesperson has been hired. Rerun the experiment to determine the average lost cash flow and the average line at closing.
5. Modify the given program so that you may calculate the average waiting time for each customer.

11
Some Applications of Your Computer

Most people don't have the time or the inclination to write programs to perform all the tasks they want their computer to perform. If you haven't done so already, you'll want to purchase programs which have been written by others. In recent years there has been a virtual explosion of commercially available programs (or, in computer jargon, software). This chapter discusses some of the "ins and outs" of purchasing such software for your computer.

11.1 Buying Software

As you probably have observed by now, it's not always easy to get a complex program up and running. To write and debug a complex program takes a considerable investment in time, wit, and dogged determination. In addition, to build the most complicated programs takes a considerable amount of technical expertise in using the various features of the computer. Most people want to use their computer to simplify various everyday tasks, but are not interested in building their major applications programs from the ground up. For these people, there's a growing collection of programs which are available through computer stores and mail order houses.

There are commercially available programs for almost every conceivable need. These programs include computer games, word processing systems, inventory control systems, appointment and record-keeping systems for professionals (doctors, dentists, lawyers), bookkeeping systems for small, medium, and large businesses, and so forth. Unfortunately, the rapid introduction of new products and the large number of available programs has made the purchase of software quite a chore. In this section, let's discuss a few pointers to help direct you through the "software jungle."

Here are some questions which you should ask yourself as part of any software purchase.

Will the program run on my computer?

In order to run a particular program, you must have the proper operating system and an adequate amount of memory.

Programs are designed to run with a particular operating system. Some models of the Franklin come with only DOS, while others are equipped with the CP/M operating system as well. The latter provides an opportunity to draw upon a huge library of programs, many of them business applications. Franklin's CP/M package includes a number of utility programs and the CBASIC language on diskettes. In any case, in buying a program, check that it will run under your operating system, and with the particular peripheral cards that you've bought and installed.

A description of a program will usually specify the amount of memory required. Obviously, you can't run a program requiring 64K of memory if you have only a 48K system! Also check to see what peripheral cards you'll need.

Will the program do what I want it to?

In purchasing a program for a particular application, you will be faced with a wide range of choices. How can you properly choose among them? Frankly, this is a real problem.

As early as possible in the software selection process, you should define your own needs as exactly as you can. To determine the extent to which a given piece of software meets those needs, you must do a fair amount of digging. Your local computer store, reviews in computer journals, and software documentation are good sources of information.

Your local computer store is the first place you should seek information. You will often be able to make a choice based on the information you get there. Many computer journals contain reviews on major pieces of software. Especially useful are reviews which compare several similar programs.

Another source of comparative information is the software documentation itself. You may often purchase the manual separately from the software. This is an expense, but it's often the only way you can be sure that the program will do exactly what you want it to do. After you have narrowed your program down to the top two contenders, why not purchase the manuals of each and inspect the two programs at close range. It's much cheaper to invest in a manual you won't need than to spend several hundred dollars on a program which will not do everything you want it to!

What do I get for my software dollar?

When you purchase software the minimum you get is the program on diskette, plus the applicable documentation. What else do you get?

1. Will the vendor accept phone calls seeking help in setting up and using the program?
2. What are the costs of updates to the software?

3. Will the vendor automatically notify you of major bugs in the software?
4. What is the quality of the documentation? Is it readily intelligible? Does it provide clear examples? Can it easily be used as a reference manual?
5. To what extent can you make required customizations of the software?
6. How much will it cost to replace defective copies of the program? Many software vendors currently supply you with only two copies of the program, which cannot be copied further. You may have to pay to replace them if they are damaged. Some programs may be copied at will from the original disk.

You should not be intimidated by the discussion above. Most software purchases proceed quite smoothly. However, as with any other area of consumer affairs, it pays to be an informed purchaser. Hopefully the suggestions above will help you spend your software dollars wisely.

11.2 Computer Communications

At some point you'll want to connect your computer to external devices (called **peripheral devices**). There are many devices available and more are being introduced at a frightening pace. At the moment, such devices include graphics screens, light pens, plotters, voice synthesizers, music synthesizers, and temperature probes, to mention only some of the possibilities. You'll want the capability of connecting your computer to other computers so that you may interchange programs and data with other users.

In this section, you'll find some of the fundamentals of computer communications. The purpose here is not to make you an expert, but to introduce you to the ideas and vocabulary so that you may read and understand the articles in various computer journals.

In most cases, it's not possible to connect two electronic devices directly to one another. It's necessary to have an intermediate device which translates the electronic signals of one device into a form intelligible to the other device. Such an intermediate device is called an **interface**; the task of electronically mating the devices is called **interfacing.** For microcomputer interfacing, there is a standard interface device called an **RS232-C** interface. The RS232-C interface allows two devices to communicate with one another using a 25-wire cable. Each of the wires carries a signal having a standardized meaning. You may purchase an RS232-C interface for your Franklin at your local computer dealer. Using this interface, you can connect your Franklin computer to a wide variety of peripheral equipment manufactured by Apple* and other outside vendors as well.

* Apple is a trademark of Apple Computer Corporation.

A word of caution: Many devices are advertised as having a built-in RS232-C interface or as being "RS232-C compatible." It may require some hard work to make them operate with your computer! There are several reasons for this: Although all RS232-C interfaces utilize a 25-wire cable, not all the wires are necessarily used. Therefore, your computer (or rather your programs) may require a signal which is not being sent, or it's not sending a signal required at the other end. A further problem lies in the confusion of connecting data sets to data terminals. There are two conventions for wiring RS232-C interfaces—one for data terminals and one for data sets. In order to connect two devices using an RS232-C interface, one must be a data terminal and one must be a data set. If both are the same variety (say both computers), then it'll be necessary to connect the interfaces on the devices by means of a special cable. The moral of all this is: When purchasing peripheral devices to connect to your computer, proceed with caution. Be sure your supplier is willing to help you or at least to exchange the device if you can't make it work.

The purpose of this section is to introduce you to some of the ideas and the vocabulary of computer communications. To begin with, let's discuss in greater detail the form in which the computer stores data.

A binary number is a string of 0's and 1's. Here is a typical example of a binary number:

```
10111011011101010000011111
```

A binary digit (that is, a 0 or a 1) is called a **bit**. A string of eight consecutive bits is called a **byte**. Here are two examples of bytes:

```
10011001 11100011
```

In the computer, all data and instructions are written in terms of binary numbers, with each byte corresponding to a character. (Except for specialized applications, don't concern yourself with the precise manner in which characters are translated into binary.) Because the basic unit of data within your machine consists of eight bits, it's called an **8-bit computer**. Larger computers (often called **main-frames**) operate with 32 or 64 bits at a time. The added efficiency thus achieved accounts, in part, for their increased speed.

There are two fundamental types of computer communications: **parallel** and **serial**. In parallel communications, a byte is transmitted all eight bits at a time. This is achieved by sending the byte over eight wires. A signal on the wire corresponds to a 1 and the absence of a signal corresponds to a 0. In serial communications, the various bits are transmitted in sequence over a single wire. Many printers utilize serial communications. In addition, serial communications are used to transmit computer data to another computer using telephone lines. The interfaces required by parallel and serial communications are quite different.

In establishing a computer communications link, there are a number of different variables which must be considered. First, there is the speed of the communications. The standard measure of communications speed is the **baud**

rate. Old-fashioned teletypes communicate at 110 baud (about 12 characters per second). Data transmission rates from your computer to a printer range from 300 to 1200 baud. High speed data transmission rates range up to 9600 baud. You may set the baud rate of your RS232-C interface using a computer command.

All communications links are subject to noise caused, primarily, by static on the lines. It's essential that computer data communications be accurate. Imagine the havoc that could be created by the erroneous transmission of a few digits of a financial report! In order to guard against such errors, many data transmission links utilize an extra bit which is tacked on to each byte. This extra bit is called a **parity bit**. The value of the parity bit depends on the sum of the other bits in the byte. It is agreed in advance whether the sum of the digits in a byte (including the parity bit) will be even (even parity) or odd (odd parity). In setting the parity bit, the computer then determines the sum of the bits in a byte. Suppose that a sum is odd and the parity is even. The parity bit will then be set to 1. The receiving device checks the parity bit to determine its correctness. If an error is detected, then a retransmission is usually requested. All this happens quite automatically. However, you must adjust your transmissions to match the parity expected. This can be done using a computer command to the RS232-C interface.

Finally, it's sometimes necessary to have a **communications protocol**. In some situations, it's useful to transmit the data at a speed higher than the receiver can accept. To do this, your computer sends the data in "bursts." Your computer can utilize the waiting time between bursts to perform other chores. At the receiving end, each burst is temporarily stored in a memory called a **buffer**. This memory holds the burst of data until the receiver has a chance to look at it. In this scheme of data transmission, it's necessary to have a pair of signals that the sender and receiver exchange. Namely, the sender must tell the receiver that more data is on the way and the receiver must tell the sender that more data may be sent. Such an exchange of signals is called a **communications protocol**. There are a number of different protocols in common use. What these protocols are is not important, but it is crucial that the sender and receiver use the **same** protocol. You may select among the most common protocols using a computer command. Note that it is necessary to use a communications protocol only in situations in which the data transmission rate is too fast for the receiver. Typically, it is not necessary to use a communications protocol with a printer at 300 baud or less. However, to get the top printing speed out of a daisy wheel printer, it's necessary to go to 1200 baud.

11.3 Information Storage and Retrieval

If your files become very large or you wish the ability to sort through them and compile complex management reports, you will need more elaborate programs than anything that has been discussed.

There are a large number of data retrieval systems which you can consider for your particular purposes. In fact, specialized programs are appearing which are structured for the needs of a particular profession (lawyer, doctor, or architect). If your accounting and information management needs go beyond those discussed (or if you don't want to bother writing your own programs), you should investigate the various packages which are commercially available.

11.4 Advanced Graphics

Computer graphics has become an incredibly sophisticated field in only a few years. Your Franklin can be used to obtain an introduction to computer graphics principles. However, you can only go so far with a black and white video display. If your Franklin has color, you should probably connect it to a color television to get a feel for the beautiful color graphics possibilities.

Many printers have a graphics mode that lets you produce hard copies of screen graphics. This is usually accomplished using "dot printing" with a high resolution of dots. To obtain even finer hard copy graphics, there are a number of plotters available which (at their most sophisticated) can faithfully produce blueprints, weather maps, and other displays.

The microcomputer user can even add a graphics tablet. This is a device that allows you to input a picture to the computer by essentially tracing the picture on a special board using an electronic "pencil." The picture is transformed into a series of dots and transmitted to the computer via a communications interface.

To survey the latest in computer devices, you should attend one of the many computer shows which take place with increasing regularity all over the country.

11.5 Connections to the Outside World

You may connect your Franklin to the outside world! To do so you must have an RS232-C interface and a special communications device called a **modem**.

A modem converts the electronic signals of your computer into signals which may be transmitted using telephone lines. A modem is connected to

your computer with the RS232-C interface. To set up a telephone connection with another computer, you first dial the number of the outside computer. Once a connection has been made, you rest the telephone receiver in the cradle of the modem. Using a direct-connect modem, you are hooked directly into a telephone line and bypass the telephone receiver entirely. (Your RS232-C should be turned on and waiting.) You have now established a communications link between you and the outside world.

You may use this communications link in many ways. First, you may communicate with other microcomputer users. You can also play games, exchange data, program ideas, and so forth. You may even use the computer as an "electronic mail service." In fact, this application of computer communications promises to revolutionize the office in the next decade. Instead of sending paper memos and printed reports, you will send such data using computer communications. If the information is to be held confidential, access will then be regulated either by passwords or encoding. Just think! No more delayed mail delivery, lost letters, or other communications problems. As a microcomputer user, you can be one of the first to use such a system.

You may use computer communications to connect your operations to a time-sharing system which you have access to. This will give you access to the greater capabilities of a larger machine as well as the program library of the time-sharing system.

Finally, you may plug your computer into any one of several information networks. Such networks normally charge a monthly fee and provide the latest stock market quotations, news, and other facts. In addition, they provide a library of programs you may use. Such information services are in their infancy and are sure to grow in number and sophistication over the next few years.

12
Where To Go From Here

This book really only scratches the surface of the computer science field and the applications in which your computer can be used. In this final chapter, let's say a few words about some of the subjects left untouched and point out some directions for further study.

12.1 Assembly Language Programming

All of your programming has been carried out in the BASIC language. There is a much more primitive language which underlies your Franklin computer, namely 6502 machine language. Actually, BASIC is **itself** a program which is written in machine language. Indeed, many complex commercial programs are written directly in machine language.

Machine language consists of the instructions which the 6502 chip can execute. These instructions tend to be much more primitive than the instructions of a higher level language such as BASIC. In a certain sense, this is unfortunate since you are forced to look at a program in very fine steps. However, the resulting programs will generally be much more efficient and will run much more quickly than programs written in BASIC. In addition, you'll understand better what is going on inside the 6502 chip in response to your instructions.

After you've become competent in BASIC, your next step can be a study of machine language. In order to help you get started, let's spend a short time discussing how machine language works.

The internal workings of the computer are all carried out in binary. This includes machine language commands. However, it's extremely difficult to write a program which is nothing but a long string of 0's and 1's. To ease this tremendous burden, you write machine language commands in terms of **mne-**

monics. These are similar to instruction designations used in BASIC. The program, written in terms of mnemonics, is called the **source code.**

The next step in preparing a machine language program is to translate mnemonics into binary. This is done using a program called an **assembler.** The resulting program is called the **object code** or **machine code.** You may list the object code but it's extremely difficult to read since it consists of an endless string of 0's and 1's. To ease this burden, computer scientists use a notational system consisting of 16 symbols, namely 0-9 and A-F. This system is called the **hexadecimal system** and may be used to list the object code of a program. Moreover, all memory addresses are specified in terms of hexadecimal notation. Because of its direct relationship to binary, hexadecimal notation is directly intelligible to the computer.

In the process of running the assembler, you must decide where in memory your program is to be stored. This is a complication that you don't worry about in BASIC programming. BASIC finds memory space and keeps track of where the various parts of the program are located. However, in machine language programming, all the internal bookkeeping is your responsibility. Once your program is assembled, you are ready to load and run it.

You might wonder if machine language programming is really worth the effort described above. Probably not in the case of a program you plan to use once or twice. However, if you are planning a program which you will be using often, perhaps as a subroutine in many different BASIC programs, it will then probably be worth the invested time to write the program in machine language. First of all, your program will run much faster. Second, you'll be able to make the screen, keyboard, and printer perform actions which may be clumsy or downright impossible to specify in BASIC.

12.2 Other Languages and Operating Systems

BASIC is only one of several hundred different computer languages. It is only one of the possible languages which are available to run on your Franklin computer. Mastering one or more of these other languages is another possible area for further study.

As microcomputers have become more common, many of the languages designed to run on large computer systems have been configured for microcomputers. Any list of available languages will probably be incomplete by the time this book goes to press. Nevertheless, let's mention some of the most common languages which are available for the Franklin computer.

The old standard of computing languages is FORTRAN. This is a powerful language especially useful in scientific and engineering applications.

FORTRAN is a **compiler**, while Floating Point BASIC is an **interpreter**. This is an important distinction. With an interpreter, you type in the program directly as it will be executed. In order to execute a particular instruction, the computer refers to a machine-language subroutine to "interpret" the intent. With a compiler, you type in the program in much the same manner as with an interpreter. However, you must **compile** the program prior to running it. That is, you must run a special routine which translates the various typed instructions into machine language. It's the machine language version of the program which you actually run. A compiled program is much more efficient than a program written with an interpreter. Depending on the program, the compiled program will run from 5 to 50 times faster!

You may supplement your BASIC interpreter with a BASIC compiler. This will allow you to program in a language you already know (once you learn the intricacies of the compiler version) and yet achieve the efficiencies of compiled programs.

COBOL and PASCAL are two very popular languages. COBOL is probably the most commonly used language for business programs. It's designed to allow ease in preparing management and financial reports. PASCAL is an extremely powerful language which may be used for general programming. It allows you to write complex programs in very few commands.

In purchasing languages for your computer, it is important to recognize that the particular version you purchase must be compatible with your operating system. In order to make use of other languages or other programs, you may wish to add another operating system. The most likely candidate is CP/M, the most common microcomputer operating system. This may be accomplished by adding an additional card in one of the empty slots of the Franklin if you haven't already done so.

Hopefully, this book has sparked your interest in microcomputing so that you'll pursue some of the suggestions for further study. Good luck!

Answers to Selected Exercises

CHAPTER 2
Section 2.2 (page 17)

1.
```
10 PRINT 57+23+48
20 END
```
2.
```
10 PRINT 57.83*(48.27 -12.54)
20 END
```
3.
```
10 PRINT 127.86/38
20 END
```
4.
```
10 PRINT 365/.005+1.02^5
20 END
```
5.
```
10 PRINT 2^1,2^2,2^3
20 PRINT 3^1,3^2,3^3
30 PRINT 4^1,4^2,4^3
40 PRINT 5^1,5^2,5^3
50 PRINT 6^1,6^2,6^3
```
6.
```
10 PRINT "CAST REMOVAL",45
20 PRINT "THERAPY",35
30 PRINT "DRUGS",5
40 PRINT ""
50 PRINT "TOTAL",45+35+5
60 PRINT "MAJ MED", .8*(45+35+5)
70 PRINT "BALANCE", .2*(45+35+5)
80 END
```
7.
```
10 PRINT "THACKER", 698+732+129+487
20 PRINT "HOVING", 148+928+246+201
30 PRINT "WEATHERBY",379+1087+148+641
40 PRINT "TOTAL VOTES", 698+732+129+487+148+928+
   246+201+379+1087+148+641
50 END
```

   Note that line 40 extends over two lines of the screen. To type such a line, just keep typing and do not hit a carriage return until you are done with the line. The maximum line length is 255 characters.

8. -2
9.
```
SILVER    GOLD    COPPER
327       449     1052
```

10.
```
          GROCERIES     MEATS
   MON      1,245         2,348
   TUE        248         3,459
```
11. 2.3E7
12. 1.7525E2
13. -2E8
14. 1.4E-4
15. -2.75E-10
16. 5.342E16
17. 159,000
18. -20,345,600
19. -.00000000000007956
20. .00000000000000000239456

Section 2.3 (page 24)

1. 10
2. 0
3. 50
4. 9 -7 18
5. JOHN JONES AGE 38
6. 22
   57
7. A can only assume numeric constants as values.
8. Nothing.
9. A$ can only assume string constants as values.
10. No line number. String constant not in quotes.
11. Nothing
12. A variable name must begin with a letter.
13.
```
10 LET A=2.3758:B=4.58321:C=58.11
20 PRINT A+B+C
30 PRINT A*B*C
40 PRINT A^2+B^2+C^2
50 END
```
14.
```
10 LET A$="Office Supplies":B$="Computers":C$="Newsletters"
20 LET RA=346712:RB=459321:RC=376872
30 LET EA=176894:EB=584837:EC=402195
40 PRINT ,A$,B$,C$
50 PRINT "REVENUE",RA,RB,RC
60 PRINT "EXPENSES",EA,EB,EC
70 LET PA=RA-EA:PB=RB-EB:PC=RC-EC
80 PRINT "PROFIT",PA,PB,PC
90 PRINT
100 PRINT "TOTAL PROFIT EQUALS",PA+PB+PB
110 END
```

Section 2.4 (page 34)

1.
```
10 LET S=0
20 FOR J=1 TO 25
30 LET S=S+J^2
40 NEXT J
50 PRINT S
60 END
```
2.
```
10 LET S=0
20 FOR J=0 TO 10
30 LET S=S+(1/2)^J
40 NEXT J
50 PRINT S
60 END
```
3.
```
10 LET S=0
20 FOR J=1 TO 10
30 LET S=S+J^3
40 NEXT J
50 PRINT S
60 END
```
4.
```
10 LET S=0
20 FOR J=1 TO 100
30 LET S=S+1/J
40 NEXT J
50 PRINT S
60 END
```
5.
```
10 PRINT "N","N^2","N^3"
20 FOR J=1 TO 12
30 PRINT J,J^2,J^3
40 NEXT J
50 END
```
6.
```
10 PRINT "MONTH","INTEREST","BALANCE"
20 B=4000,P=125.33
30 FOR J=1 TO 12
40 LET I=.01B:REM I=THE INTEREST FOR MONTH
50 LET R=P-I:REM R=REDUCTION IN BALANCE FOR MONTH
60 LET B=B-R:REM NEW BALANCE
70 PRINT J,I,B
80 NEXT J
90 END
```
7.
```
10 PRINT "END OF YEAR", "BALANCE"
20 B=1000
30 FOR J=1 TO 15
40 B=B+1000+.10*B :REM ADD DEPOSIT AND INTEREST
50 PRINT J,B
60 NEXT J
70 END
```

8.
```
10 LET S=3.5E7: P=5.54E6
20 PRINT "END OF YEAR", "SALES", "PROFITS"
30 FOR J=1 TO 3
40 LET S=1.2*S: P=1.3*P
50 PRINT J,S,P
60 NEXT J
70 END
```

Section 2.6 (page 46)

1.
```
10 J=1
20 IF J^2 > 45000 THEN 100
30 PRINT J,J^2
40 J=J+1
50 GOTO 20
100 END
```
2.
```
10 PI=3.14159
20 R=1
30 IF PI*R^2 <= 5000 THEN 40
35 GOTO 100
40 PRINT R,PI*R^2
50 R=R+1
60 GOTO 30
100 END
```
3.
```
10 PRINT "SIDE OF CUBE","VOLUME"
20 S=1
30 V=S^3
40 IF V <175000 THEN 50
45 GOTO 100
50 PRINT S,V
60 S=S+1
70 GOTO 30
100 END
```
4.
```
10 FOR J=1 TO 10 : REM LOOP TO GIVE 10 PROBLEMS
20 INPUT "TYPE TWO 2-DIGIT NUMBERS"; A,B
30 INPUT "WHAT IS THEIR PRODUCT";C
40 IF A * B=C THEN 200
50 PRINT "SORRY, THE CORRECT ANSWER IS",A*B
60 GOTO 500 : REM GO TO THE NEXT PROBLEM
200 PRINT "YOUR ANSWER IS CORRECT! CONGRATULATIONS"
210 LET R=R+1 : REM INCREASE SCORE BY 1
220 GOTO 500 : REM GO TO THE NEXT PROBLEM
500 NEXT J
600 PRINT "YOUR SCORE IS",R,"CORRECT OUT OF 10"
700 PRINT "TO TRY AGAIN, TYPE RUN"
800 END
```
5.
```
10 FOR J=1 TO 10 : REM LOOP TO GIVE 10 PROBLEMS
15 PRINT "CHOOSE OPERATION TO BE TESTED:"
```

```
16 PRINT "ADDITION (A), SUBTRACTION (S), OR MULTIPLICATION
   (M)"
17 INPUT A$
20 INPUT "TYPE TWO 2-DIGIT NUMBERS:"; A,B
21 IF A$="A" THEN 30
22 IF A$="S" THEN 130
23 IF A$="M" THEN 230
24 GOTO 15
30 INPUT "WHAT IS THEIR SUM";C
40 IF A+B=C THEN 400
50 PRINT "SORRY. THE CORRECT ANSWER IS",A+B
60 GOTO 500 : REM GO TO THE NEXT PROBLEM
130 INPUT "WHAT IS THEIR DIFFERENCE";C
140 IF A-B=C THEN 400
150 PRINT "SORRY. THE CORRECT ANSWER IS",A-B
160 GOTO 500 : REM GO TO THE NEXT PROBLEM
230 INPUT "WHAT IS THEIR PRODUCT";C
240 IF A * B=C THEN 400
250 PRINT "SORRY. THE CORRECT ANSWER IS",A*B
260 GOTO 500 : REM GO TO THE NEXT PROBLEM
400 PRINT "YOUR ANSWER IS CORRECT! CONGRATULATIONS"
410 LET R=R+1 : REM INCREASE SCORE BY 1
420 GOTO 500 : REM GO TO THE NEXT PROBLEM
500 NEXT J
600 PRINT "YOUR SCORE IS",R,"CORRECT OUT OF 10"
700 PRINT "TO TRY AGAIN, TYPE RUN"
800 END
```

6. See Exercise 8.
7. See Exercise 9.
8.
```
10 INPUT "NUMBER OF NUMBERS:";N
20 FOR J=1 TO N
30 INPUT A
40 IF J=1 THEN B=A
50 IF A>B THEN B=A
60 NEXT J
70 PRINT "THE LARGEST NUMBER INPUT IS","B
80 END
```
9. Replace line 50 in Exercise 8. by:
```
50 IF A<B THEN B=A
```
10.
```
10 A0=5782:A1=6548:B0=4811:B1=6129:C0=3865:C1=4270
20 D0=7950:D1=8137:E0=4781:E1=4248:F0=6598:F1=7048
30 FOR J=1 TO 6
40 IF J=1 THEN A=A0:B=A1
50 IF J=2 THEN A=B0:B=B1
60 IF J=3 THEN A=C0:B=C1
70 IF J=4 THEN A=D0:B=D1
80 IF J=5 THEN A=E0:B=E1
```

```
90 IF J=6 THEN A=F0:B=F1
100 I=B-A
110 IF I>0 THEN PRINT "CITY ";J;" HAD AN INCREASE OF "; I
120 GOTO 200
130 IF I<0 THEN PRINT "CITY ";J;" HAD A DECREASE OF ";A-B
140 GOTO 300
200 IF I>500 THEN PRINT "CITY ";J;" MORE THAN 500 INCREASE"
300 NEXT J
400 END
```

11.
```
10 PRINT "THIS PROGRAM SIMULATES A CASH REGISTER"
20 PRINT "AT THE QUESTION MARKS, TYPE IN THE PURCHASE"
30 PRINT "AMT'S. TYPE -1 TO INDICATE THE END OF ORDER"
40 INPUT "TYPE 'Y' IF READY TO BEGIN"; A$
50 IF A$="Y" THEN 60
55 GOTO 10
60 HOME
70 INPUT"ITEM "; A
80 IF A=-1 THEN 200
85 GOTO 90
90 T=T+A: REM T IS THE RUNNING TOTAL
100 GOTO 70
200 PRINT "THE TOTAL IS", T
210 S=.05*T:REM S=SALES TAX
220 PRINT "SALES TAX", S
230 PRINT "TOTAL DUE", S+T
240 INPUT "PAYMENT GIVEN";P
250 PRINT "CHANGE DUE", P-(S+T)
300 END
```

12.
```
10 INPUT "CASH ON HAND:"; C1
20 PRINT "INPUT ACCOUNTS EXPECTED TO BE RECEIVED IN NEXT
   MONTH."
30 PRINT "TO INDICATE END OF ACCOUNTS TYPE -1."
40 INPUT "ACCOUNTS RECEIVABLE";A
50 IF A=-1 THEN 100
60 C2=C2+A:REM C2=RUNNING TOTAL OF ACCOUNTS RECEIVABLE
70 GOTO 40
100 PRINT "INPUT ACCOUNTS EXPECTED TO BE PAID IN NEXT
    MONTH."
110 PRINT "TO INDICATE END OF ACCOUNTS TYPE -1."
120 INPUT "ACCOUNTS PAYABLE";A
130 IF A=-1 THEN 200
140 C3=C3+A:REM C3=RUNNING TOTAL OF ACCOUNTS PAYABLE
150 GOTO 120
200 PRINT "CASH ON HAND",C1
220 PRINT "ACCOUNTS RECEIVABLE",C2
230 PRINT "ACCOUNTS PAYABLE",C3
240 PRINT "NET CASH FLOW", C1+C2-C3
300 END
```

CHAPTER 3
Section 3.1 (page 58)

1. DIM A(5)
2. DIM A(2,3)
3. DIM A$(3)
4. DIM A(3)
5. DIM A$(4),B(4)
6. 10 DIM A$(3),B(3,2),C$(2)
   20 PRINT ,"Receipts"
   30 C$(1)="STORE #1":C$(2)="STORE #2"
   40 A$(1)="1/1-1/10":A$(2)="1/11-1/20":A$(3)="1/21-1/31"
   50 B(1,1)=57385.48:B(1,2)=89485.45
   60 B(2,1)=39485.98:B(2,2)=76485.49
   70 B(3,1)=45467.21:B(3,2)=71494.25
   100 PRINT ,C$(1),C$(2)
   200 FOR J=1 TO 3
   220 PRINT A$(J),B(J,1),B(J,2)
   230 NEXT J
   300 END
7. Add the instructions:
   5 DIM D(2)
   240 FOR J=1 TO 2
   250 D(J)=B(1,J)+B(2,J)+B(3,J)
   260 NEXT J
   270 PRINT "TOTALS",D(1),D(2)
8. Move the END to 400 and add the following instructions.
   6 DIM E(3)
   300 FOR J=1 TO 3
   310 E(J)=B(J,1)+B(J,2)
   320 NEXT J
   330 PRINT
   340 PRINT "PERIOD","TOTAL SALES"
   350 FOR J=1 TO 3
   360 PRINT A$(J),E(J)
   370 NEXT J
   400 END
9. 10 DIM A$(4), B$(5), C(5,4)
   20 A$(1)= "STORE #1":A$(2)="STORE #2": A$(3)="STORE #3"
   21 A$(4)= "STORE #4"
   30 B$(1)= "REFRIG.":B$(2)="STOVE":B$(3)="VACUUM"
   40 B$(4)= "AIR COND": B$(5)="DISPOSAL"
   50 PRINT "INPUT THE CURRENT INVENTORY"
   60 FOR J=1 TO 4
   70 PRINT A$(J)
   80 PRINT
   90 FOR I=1 TO 5

```
100 PRINT B$(I)
110 INPUT C(I,J)
120 NEXT I
130 NEXT J
200 REM REST OF PROGRAM IS FOR INVENTORY UPDATE
210 PRINT "CHOOSE ONE OF THE FOLLOWING"
220 PRINT "RECORD SHIPMENTS(R)"
230 PRINT "DISPLAY CURRENT INVENTORY(D)"
240 INPUT "TYPE R OR D";D$
250 IF D$= "R" THEN 300
260 IF D$= "D" THEN 600
270 HOME:GOTO 200
300 HOME
310 PRINT "RECORD SHIPMENT"
320 INPUT "TYPE STORE#(1-4)";J
330 PRINT "ITEM SHIPPED"
340 PRINT "REFRIG=1,STOVE=2,VACUUM=3,AIR COND=4,DISPOSAL=5"
350 INPUT I
360 INPUT "NUMBER SHIPPED";S
370 C(I,J)=C(I,J)-S
380 GOTO 200
600 HOME
620 FOR J=1 TO 5
630 FOR I=1 TO 4
640 PRINT A$(I);B$(J),C(J,I)
650 NEXT I
660 NEXT J
670 GOTO 200
1000 END
```

Note that this program is really an infinite loop. For this type of program this is a good idea. You don't want to accidentally end the program thereby erasing the current inventory figures! End this program using the **CTRL C** key combination.

Section 3.2 (page 64)

1. `A(1)=2,A(2)=4,A(3)=6,A(4)=8,A(5)=10,A(6)=12,A(7)=14,A(7)=16,A(8)=18,A(9)=20`
2. `A(0)=1.1,A(1)=3.3,A(2)=5.5,A(3)=7.7, B(0)=2.2, B(1)=4.4, B(2)=6.6, B(3)=8.8`
3. `A(0)=1, A(1)=2, A(2)=3, A(3)=4, B$(0)="A", B$(1)="B", B$(2)="C", B$(3)="D"`
4. `A(0)=1,B(0)=2,A(1)=3,B(1)=4,A(2)=1,B(2)=2,A(3)=3,B(3)=4`
5. `A(1,1)=1,A(1,2)=2,A(1,3)=3,A(1,4)=4,A(2,1)=5,A(2,2)=6,A(2,3)=7,A(2,4)=8,A(3,1)=9,A(3,2)=10,A(3,3)=11,A(3,4)=12`

6. A(1,1)=1,A(2,1)=2,A(3,1)=3,A(1,2)=4,A(2,2)=5,A(3,2)=6,A(1,3)=7,A(2,3)=8,A(3,3)=9,A(1,4)=10,A(2,4)=11,A(3,4)=12
7. Out of DATA in 30.
8. Type Mismatch in 30. (Attempt to set numeric variable equal to string.)
9. Set F(J) equal to the Federal withholding for employee J, N(J) = the net pay, and add the following lines.

```
280 PRINT "EMPLOYEE", "WITHHOLDING","NET PAY"
290 FOR J=1 TO 5
300 IF D(J) <= 200 THEN F(J)=0:GOTO 500
310 IF D(J) <= 210 THEN F(J) =29.10:GOTO 500
320 IF D(J) <= 220 THEN F(J)=31.20:GOTO 500
330 IF D(J) <= 230 THEN F(J)=31.80:GOTO 500
340 IF D(J) <= 240 THEN F(J)=36.40:GOTO 500
350 IF D(J) <= 250 THEN F(J)=39.00:GOTO 500
360 IF D(J) <= 260 THEN F(J)=41.60:GOTO 500
370 IF D(J) <= 270 THEN F(J)=44.20:GOTO 500
380 IF D(J) <= 280 THEN F(J)=46.80:GOTO 500
390 IF D(J) <= 290 THEN F(J)=49.40:GOTO 500
400 IF D(J) <= 300 THEN F(J)=52.10:GOTO 500
410 IF D(J) <= 310 THEN F(J)=55.10:GOTO 500
420 IF D(J) <= 320 THEN F(J)=58.10:GOTO 500
430 IF D(J) <= 330 THEN F(J)=61.10:GOTO 500
440 IF D(J) <= 340 THEN F(J)=64.10:GOTO 500
450 IF D(J) <= 350 THEN F(J)=67.10:GOTO 500
500 N(J)=D(J)-E(J)-F(J)
600 PRINT B$(J),F(J),N(J)
700 NEXT J
```

10.

```
5 DIM A(25)
10 DATA 10,10,9,9,8,11,15,18,20,25,31,35,38,39,40,40,42,38
20 DATA 33,27,22,18,15,12
30 FOR J=0 TO 23
40 READ A(J)
50 S=S+A(J)
60 NEXT J
70 PRINT "AVERAGE 24 HOUR TEMP.", S/24
100 PRINT "TO FIND THE TEMPERATURE AT ANY PARTICULAR HOUR"
110 PRINT "TYPE THE HOUR IN 24-HOUR NOTATION: 0-12=AM"
120 PRINT "13-23=PM"
130 PRINT "TO END THE PROGRAM, TYPE 25"
140 INPUT "DESIRED HOUR";A
150 IF A=25 THEN 200
160 PRINT "THE QUERIED TEMPERATURE WAS";A(A);"DEGREES"
170 GOTO 100
200 END
```

Section 3.3 (page 69)

1. 10 PRINT "THE VALUE OF X IS";5.378
   20 END
2. 10 PRINT "THE VALUE OF X IS";TAB(23) 5.378
   20 END
3. 10 PRINT "DATE";TAB(6)"QTY";TAB(12)"@";TAB(17)"COST";
   20 PRINT TAB(25) "DISCOUNT";TAB(35)"COST"
   30 END
4. 10 X=6.753:Y=15.111:Z=111.850:W=6.702
   20 PRINT SPC(7-LEN(STR$(X));X
   30 PRINT SPC(7-LEN(STR$(Y));Y
   40 PRINT SPC(7-LEN(STR$(Z));Z
   50 PRINT SPC(7-LEN(STR$(W));W
   60 PRINT "-------"
   70 S=X+Y+Z+W
   80 PRINT SPC(7-LEN(STR$(S));S
   90 END
5. 10 X=12.82:Y=117.58:Z=5.87:W=.99
   20 PRINT "$";SPC(6-LEN(STR$(X));X
   30 PRINT "$";SPC(6-LEN(STR$(Y));Y
   40 PRINT "$";SPC(6-LEN(STR$(Z));Z
   50 PRINT "$";SPC(6-LEN(STR$(W));W
   60 PRINT "-------"
   70 PRINT "$";SPC(6-LEN(STR$(X+Y+Z+W));X+Y+Z+W
   80 END
6. 10 PRINT TAB(20) "DATE";TAB(30)"3/18/81"
   20 PRINT
   30 PRINT "Pay to the Order of Wildcatters, Inc."
   50 PRINT
   60 PRINT "The Sum of";TAB(14)"**********$89,385.00."
9. 10 INPUT "NUMBER TO BE ROUNDED";X
   20 PRINT INT(X+.5)
   30 END
10. Modify the program of Exercise 11 of Section 2.6 (page 47) by substituting SPC(11-LEN(STR$(X)) into the PRINT statements. This will allow your cash register to handle numbers up to $999,999.99.

Section 3.4 (page 76)

1. 100*RND(1)
2. 100+RND(1)
3. INT(50*RND(1)+1)
4. INT(4+77*RND(1))
5. 2*INT(25*RND(1)+1)
6. 50+50*RND(1)
7. 3*INT(9*RND(1)+1)
8. 1+3*INT(7*RND(1)+1)

10. Add the following instructions:

```
132 IF C(J) > A(J) THEN 135
133 GOTO 140
135 PRINT "BET INVALID:NOT ENOUGH CHIPS PURCHASED"
137 C(J)=0
139 GOTO 120
```

11. Change line 132 in Exercise 10 to read:

```
132 IF C(J) > A(J)+100 THEN 135
133 GOTO 140
```

12.

```
10 PRINT "CHOOSE OPERATION TO BE TESTED"
20 PRINT "ADDITION(A),SUBTRACTION(S),MULTIPLICATION(M)"
30 INPUT A$
40 A=INT(10*RND(1)):B=INT(10*RND(1))
50 IF A$="A" THEN 100
60 IF A$= "S" THEN 200
70 IF A$= "M" THEN 300
100 HOME
110 PRINT "WHAT IS";A;"+";B;"?"
120 INPUT C
130 D=A+B
140 GOTO 400
200 HOME
210 PRINT "WHAT IS";A;"-";B;"?"
220 INPUT C
230 D=A-B
240 GOTO 400
300 HOME
310 PRINT "WHAT IS";A;"X";B;"?"
320 INPUT C
330 D=A*B
340 GOTO 400
400 IF C=D THEN 410
405 GOTO 420
410 PRINT "YOUR ANSWER IS CORRECT"
415 GOTO 430
420 PRINT "INCORRECT. THE CORRECT ANSWER IS", D
430 INPUT "ANOTHER PROBLEM(Y/N)";B$
440 IF B$= "Y" THEN 10
450 END
```

13. Put your names in a series of DATA statements located in lines 1000-1010

```
5 DIM A$(10)
10 FOR J=1 TO 10
20 READ A$(J)
30 NEXT J
40 FOR J=1 TO 4
50 PRINT A$(10*RND(1))
```

```
60 NEXT J
70 END
```

Section 3.5 (page 82)

1.
```
10 FOR J=.1 TO .5 STEP .1
20 GOSUB 100
30 PRINT X
40 NEXT J
50 END
100 X=5*J^2 - 3*J
110 RETURN
```
2.
```
1000 C(J)=100*(B(J)-A(J))/A(J)
1010 RETURN
```
3.
```
2000 M=C(1)
2010 FOR J=2 TO 6
2020 IF C(J)>M THEN M=C(J)
2030 NEXT J
2040 K=1
2050 IF M=C(K) THEN 2100
2055 GOTO 2060
2060 K=K+1
2070 GOTO 2050
2100 RETURN
```
5. Let D(J) = 4 mean that J bets on first 12, D(J) = 5 that J bets on second 12, D(J) = 6 that J bets on third 12. In all such bets B(J) will be 0. Corresponding to the new values of D(J), there will be three new subroutines, starting in lines 4000, 5000, and 6000, respectively. Modify lines 121 to 125 as follows:
```
121 PRINT "BET TYPE:1=NUMBER BET,2=EVEN,3=ODD,4=1st 12"
122 PRINT "5=2nd 12, 6=3rd 12"
123 INPUT "BET TYPE(1-6)";D(J)
124 IF D(J) >1 THEN 126
125 GOTO 130
126 INPUT "AMOUNT";C(J)
127 GOTO 140
```
Replace lines 320-335 inclusive by:
```
320 ON D(J) GOSUB 1000,2000,3000,4000,5000,6000
```
Finally, here are the three new subroutines.
```
4000 FOR K=1 TO 12
4010 IF X=K THEN 4100
4015 NEXT K
4020 PRINT "PLAYER";J;"LOSES"
4030 A(J)=A(J)-C(J)
4050 RETURN
4100 PRINT "PLAYER";J;"WINS";2*C(J);"DOLLARS"
4110 A(J)=A(J)+2*C(J)
4120 RETURN
```

The subroutines in 5000 and 6000 are identical, except for the lines:

```
5000 FOR K=13 TO 24
6000 FOR K=25 TO 36
```

CHAPTER 4
Section 4.2 (page 91)

1.)

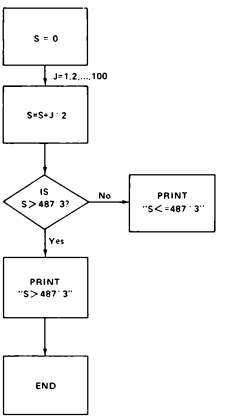


Figure A-1.

2.)

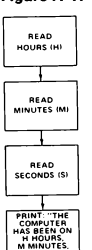


Figure A-2.

3.)

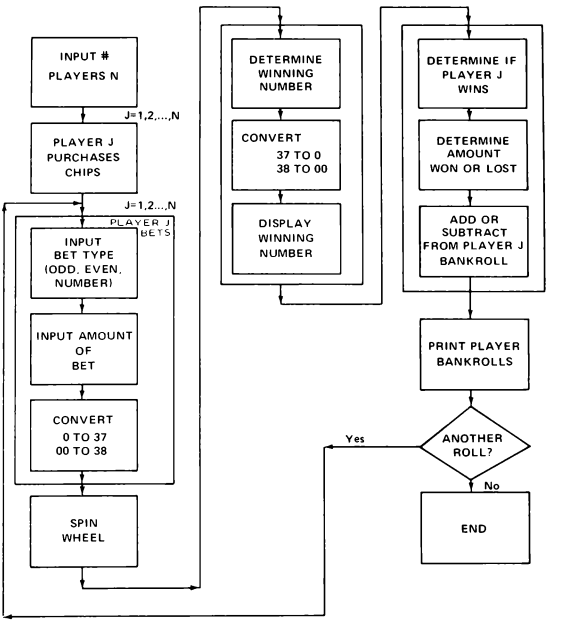

INPUT #
PLAYERS N
J=1,2,...,N
PLAYER J
PURCHASES
CHIPS
J=1,2,...,N
PLAYER J
BETS
INPUT
BET TYPE
(ODD, EVEN,
NUMBER)
INPUT AMOUNT
OF
BET
CONVERT
0 TO 37
00 TO 38
SPIN
WHEEL
DETERMINE
WINNING
NUMBER
CONVERT
37 TO 0
38 TO 00
DISPLAY
WINNING
NUMBER
DETERMINE IF
PLAYER J
WINS
DETERMINE
AMOUNT
WON OR LOST
ADD OR
SUBTRACT
FROM PLAYER J
BANKROLL
PRINT PLAYER
BANKROLLS
ANOTHER
ROLL?
Yes
No
END

Figure A-3.

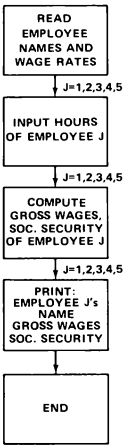


Figure A-4.

Section 4.3 (page 94)

1. Here are the errors:
   TYPE MISMATCH in line 10: "0" should be 0
   line 30: J(2 should be J^2.
   line 40: NEXT K should be NEXT J
   line 80: NXT T should be NEXT J
   line 90: should be deleted
   line 100: ST should be S*T
   line 110 quotes around "THE ANSWER IS"
2. line 30 should read: PRINT "THE FIRST N EQUALS",N
   Need line 40: GOTO 200
   line 20: IF N^2>175263 THEN 100
   should be IF N^2<175263 THEN 100
   line 110: GOTO 10 should be GOTO 20

CHAPTER 5
Section 5.3 (page 104)

1. a.
```
10 S=0
20 FOR J=1 TO 50
```

```
30 S=S+J^2
40 NEXT J
50 PRINT S
60 END
```

b. Type SAVE SQUARES

2. Type LOAD SQUARES
3. Type DELETE SQUARES

Section 5.4 (page 112)

1.
```
10 DATA 5.7,-11.4,123,485,49
20 FOR J=1 TO 5
30 READ A(J)
40 NEXT J
50 PRINT CHR$(4); "OPEN NUMBERS"
55 PRINT CHR$(4); "WRITE NUMBERS"
60 FOR J=1 TO 5
70 PRINT A(J)
80 NEXT J
90 PRINT CHR$(4); "CLOSE NUMBERS"
100 END
```

2.
```
10 PRINT CHR$(4); "OPEN NUMBERS"
15 PRINT CHR$(4); "READ NUMBERS"
20 FOR J=1 TO 5
30 INPUT A(J)
40 NEXT J
50 PRINT CHR$(4); "CLOSE NUMBERS"
60 FOR J=1 TO 5
70 PRINT A(J)
80 NEXT J
100 END
```

3.
```
5 PRINT CHR$(4); "OPEN NUMBERS"
10 PRINT CHR$(4); "APPEND NUMBERS"
15 PRINT CHR$(4); "WRITE NUMBERS"
20 DATA 5, 78, 4.79, -127
30 FOR J=1 TO 4
40 READ A
45 PRINT A
50 NEXT J
60 PRINT CHR$(4); "CLOSE NUMBERS"
70 END
```

4.
```
5 PRINT CHR$(4); "OPEN NUMBERS"
10 PRINT CHR$(4); "READ NUMBERS"
20 FOR J=1 TO 9
30 INPUT A(J)
40 NEXT J
50 PRINT CHR$(4); "CLOSE NUMBERS"
60 FOR J=1 TO 9
```

```
70 PRINT A(J)
80 NEXT J
100 END
```

5.
```
5 J=0
10 DIM A(100),B$(100),C$(100),D(100),E$(100)
20 PRINT "TYPE CHECK DATA ITEMS REQUESTED."
30 PRINT "FOLLOW EACH ITEM BY A CARRIAGE RETURN."
40 J=J+1
50 INPUT "CHECK #";A(J)
60 INPUT "DATE";B$(J)
70 INPUT "PAYEE";C$(J)
80 INPUT "AMOUNT(NO. $)";D(J)
90 INPUT "EXPLANATION";E$(J)
100 INPUT "ANOTHER CHECK(Y/N)";F$
120 HOME
130 IF F$= "Y" THEN 20
200 PRINT CHR$(4); "OPEN CHECKS"
210 PRINT CHR$(4); "WRITE CHECKS"
220 FOR M=1 TO J
230 PRINT A(M),B$(M),C$(M),D(M),E$(M)
240 NEXT M
250 PRINT CHR$(4); "CLOSE CHECKS"
1000 END
```

6.
```
10 PRINT CHR$(4); "OPEN CHECKS"
15 PRINT CHR$(4); "READ CHECKS"
20 ONERR GOTO 100
30 J=1
40 INPUT A(J),B$(J),C$(J),D(J),E$(J)
50 J=J+1
60 GOTO 40
100 PRINT CHR$(4); "CLOSE CHECKS"
110 S=0
120 FOR M=1 TO J
130 S=S+D(M)
140 NEXT M
150 PRINT "TOTAL OF CHECKS IS",S
1000 END
```

CHAPTER 6
Section 6.1 (page 118)

1.
```
5 GR:COLOR=1
10 HLIN 0,39 AT 18
```

2.
```
5 GR: COLOR=1
10 VLIN 0,39 AT 17
```

3.
```
5 GR:COLOR=1
10 HLINE 0,39 AT 19
20 VLIN 0,39 AT 19
```

4.
```
5 GR:COLOR=1
10 HLIN 0,39 AT 13
20 HLIN 0,39 AT 26
30 VLIN 0,39 AT 13
40 VLIN 0,39 AT 26
```
5.
```
5 GR: COLOR=1
10 VLIN 1,24 AT 30
20 VLIN 1,24 AT 31
```
6.
```
5 GR: COLOR=1
10 FOR J=0 TO 39
20 PLOT J,J
30 NEXT J
40 END
```
7.
```
5 GR: COLOR=1
10 FOR J=1 TO 39
20 PLOT J,12
30 NEXT J
40 FOR K=1 TO 3
50 PLOT 10*K,13
60 PLOT 10*K,11
70 NEXT K
80 END
```
8.
```
5 GR: COLOR=1
10 FOR J=1 TO 39
20 PLOT 19,J
30 NEXT J
40 FOR J=0 TO 4
50 PLOT 18,8*J
55 PLOT 20,8*J
60 NEXT J
70 END
```

Section 6.3 (page 121)

2. The entire screen will be full.

CHAPTER 7

Section 7.2 (page 124)

3.
```
10 A$= "15+48+97=160"
20 B$(1)=LEFT$(A$,2)
30 B$(2)=MID$(A$,4,2)
40 B$(3)=MID$"(A$,7,2)
50 B$(4)=RIGHT$(A$,3)
60 FOR J=1 TO 4
70 B(J)=VAL(B$(J))
80 NEXT J
90 FOR J=1 10 3
100 PRINT SPC(3-LEN(STR$(B(J)));B(J)
```

```
110 NEXT J
120 PRINT "---"
130 PRINT SPC(3-LEN(STR$(B(J)))); B(4)
140 END
```

4.
```
10 A$="$6718.49": B$="$4801.96"
20 A1$= RIGHT$(A$,7):B1$= RIGHT$(B$,7)
30 A2=VAL(A1$): B2=VAL(B1$)
40 PRINT "$";SPC(9-LEN(A1$));A2
50 PRINT "$";SPC(9-LEN(B1$));B2
60 PRINT "----------"
70 PRINT"$";SPC(9-LEN(STR$(A2+B2)));  A2+B2
80 END
```

CHAPTER 9

Section 9.1 (page 163)

1. 3.0000000000
2. 2.3700000000
3. 578,000.0000
4. 2.0000000000
5. 3.0000000000
6. -4.1000000000
7. -4
8. 3500.685000
9. 217.6000000
10. -5,940,000,000,000
11. 3.5869500400
12. -2.34542383E10
13. -236,700,000,000,000,000,000
14. 4570000000000000000
15. 46.00000000
16. .5000000000
17. .6000000000
18. 1.600000000
19. .6666666667
20. 1.1966666667
21. 4963
22. 1749.999999

Section 9.2 (page 164)

1.
```
10 PRINT (5.87+3.85 - 12.07)/11.98
20 END
```
2.
```
10 PRINT (15.1+11.9)^4/12.88
20 END
```
3.
```
10 PRINT (32485+9826)/(321.5 - 87.6^2)
20 END
```
4.
```
10 INPUT X%
20 IF X% < 0 THEN X%=X%-1
```

```
   30 PRINT X%
   40 END
```
5. -5
6. 4
7. -12
8. 1.7800000000
9. .0010000000
10. 32.65342000
11. 4.252345443E21
12. -1.234567890E-32
13. 3.283646493
14. -5.740000000

Section 9.3 (page 169)

1.
```
   10 PRINT EXP(1.54)
   20 END
```
2.
```
   10 PRINT EXP(-2.376)
   20 END
```
3.
```
   10 PRINT LOG(58)
   20 END
```
4.
```
   10 PRINT LOG(9.75E-5)
   20 END
```
5.
```
   10 PRINT SIN(3.7)
   20 END
```
6.
```
   10 PRINT COS(.017453*45)
   20 END
```
7.
```
   10 PRINT ATN(1)
   20 END
```
8.
```
   10 PRINT TAN(.682)
   20 END
```
9.
```
   10 PRINT 57.29578*ATN(2)
   20 END
```
10.
```
   10 PRINT LOG(18.9)/LOG(10)
   20 END
```
11.
```
   10 FOR X=-5.0 TO 5.0 STEP .1
   20 PRINT X, EXP(X)
   30 NEXT X
   40 END
```
12.
```
   10 DATA 1.7, 3.1, 5.9, 7.8, 8.4, 10.1
   20 FOR J=1 TO 6
   30 READ X
   40 PRINT X, 3*X^(1/4)*LOG(5*X)+EXP(-1.8*X)*TAN(X)
   50 NEXT J
   60 END
```
15.
```
   10 INPUT X
   20 PRINT "THE FRACTIONAL PART OF",X,"IS", X-INT(X)
   30 END
```

Section 9.4 (page 170)

1. `10 DEF FNA(X)=X^2-5*X`
2. `10 DEF FNA(X)=1/X-3*X`
3. `10 DEF FNA(X)=5*EXP(-2*X)`
4. `10 DEF FNA(X)=X*LOG(X/2)`
5. `10 DEF FNA(X)=TAN(X)/X`
6. `10 DEF FNA(X)=COS(2*X)+1`
7. ```
   10 DEF FNA(X)=5*EXP(-2*X)
   20 FOR X=0 TO 10 STEP .1
   30 PRINT X, FNA(X)
   40 NEXT X
   50 END
   ```
   ```

INDEX

$14.95

ISBN: 0-89303-341-3